

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA INFORMÀTICA
Grado en Ingeniería Informática



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Escola Tècnica
Superior d'Enginyeria
Informàtica



etsinf

ST-LIB: LIBRERÍA DE ABSTRACCIÓN DE
MICROCONTROLADORES STM32

TRABAJO DE FIN DE GRADO

Autor:

Ricardo Chust Fides

Tutor:

Vicente Atienza

Colaboradores:

**Daniel González, Stefan Costea, Alejandro
Gonzalvo, Pablo González**

VALÈNCIA, 2023

Resumen

Este TFG tratará de documentar tanto el desarrollo como la funcionalidad de la librería ST-LIB para microcontroladores STM32, una librería desarrollada para la competición Hyperloop Week por parte de Hyperloop UPV. El propósito de la librería es abstraer lo máximo posible la implementación del código para las placas que usen microcontroladores STM32, y así reducir notablemente el tiempo y coste de desarrollo de infraestructuras que usen estas placas.

Por su parte, el propósito de este documento será servir como herramienta a generaciones futuras de Hyperloop y a potenciales usuarios externos para poder entender en profundidad los propósitos de la librería y sus capacidades, así como aprender a utilizarla rápidamente y tener un referente para futuros proyectos similares.

Dentro del documento se divide en dos grandes partes, una orientada al desarrollo del software de la librería y otra parte a las capacidades funcionales que ofrece como herramienta para ingeniería de computadores. La primera parte tratará las metodologías de desarrollo empleadas, la gestión del reparto de trabajo, las decisiones tomadas mientras se construía y sus posibles casos de uso fuera del equipo Hyperloop UPV. La segunda parte documentará la librería para facilitar su aprendizaje, analizará sus capacidades funcionales, discutirá posibles expansiones futuras a esta y ofrecerá un análisis de sus costes temporales y espaciales.

Palabras clave: ST-LIB, STM32, Hyperloop, ingeniería de computadores, desarrollo de software, microcontrolador

Abstract

This TFG will try to document both the development and the functionality of the ST-LIB library for STM32 microcontrollers, a library developed for the Hyperloop Week competition by Hyperloop UPV. The purpose of the library is to abstract as much as possible the implementation of the code for boards that use STM32 microcontrollers, and thus significantly reduce the time and cost of developing infrastructures that use these boards.

For its part, the purpose of this document will be to serve as a tool for future generations of Hyperloop and potential external users to understand in depth the purposes of the library and its capabilities, as well as learn to use it quickly and have a reference for future similar projects.

Within the document there is a division between two large parts, one oriented to the development of the library software and another part to the functional capabilities that it offers as a tool for computer engineering. The first part will deal with the development methodologies used, the management of the distribution of work, the decisions made while it was being built and its possible use cases outside the Hyperloop UPV team. The second part will document the library to facilitate your learning, analyze its functional capabilities, discuss possible future expansions to it, and offer an analysis of its temporal and spatial costs.

Key words: ST-LIB, STM32, Hyperloop, computer engineering, software development, microcontroller

Índice general

1	Glosario de términos	3
2	Formato del documento	6
3	Objetivo del proyecto	7
3.1	EHW	7
3.2	Uso externo	8
3.3	Estado del arte	8
4	instalación y uso	10
4.1	Git	10
4.2	IDE	11
4.3	Comenzando a usar la librería	14
4.3.1	Activar y desactivar módulos	14
4.3.2	Configurar Pins y Runes	14
4.3.3	Prueba básica de funcionamiento	18
5	Desarrollo de la librería	20
5.1	Métodos de gestión de proyecto	20
5.2	Gitflow	21
6	Diseño y funcionalidad	25
6.1	ST-LIB Core	25
6.1.1	Modelos y Servicios	25
6.1.2	Pin	26
6.1.3	DMA	27
6.1.4	Paquetes	28
6.1.5	Periféricos básicos: Analog y Digital	29
6.1.6	PWM	31
6.1.7	Time	32
6.1.8	CORDIC	33
6.1.9	Encoder	35
6.1.10	Flash	36
6.2	ST-LIB Communication	37
6.2.1	UART	37
6.2.2	I2C y SPI	39
6.2.3	CAN-FD	41
6.2.4	UDP	42
6.2.5	TCP	44
6.2.6	SNTP	46
6.3	ST-LIB Low	47
6.3.1	Counter	47
6.3.2	StopWatch	48
6.3.3	DigitalOutput	49

6.3.4	ErrorHandler	49
6.3.5	Math	50
6.3.6	Sensors	51
6.3.7	EncoderSensor	53
6.3.8	StateMachine	56
6.4	ST-LIB High	59
6.4.1	Notification	59
6.4.2	ProtectionManager	60
7	Testeo y perfilado de la librería	62
7.1	Perfilado	62
7.2	En la práctica	65
7.3	Testing automático	68
8	Discusión del proyecto	71
9	Conclusión	73

1. Glosario de términos

anidar Convocar llamadas a funciones o contener estructuras de datos dentro de otras funciones o estructuras.

array Vector de valores de tamaño fijo.

ATP Automatic Testing Platform. Un pequeño sistema de revisión de *Git* que comprueba si los cambios aplicados a la librería funcionan correctamente.

branching Término usado dentro de Git para referirse a la creación masiva de ramas.

Capa En referencia al código, agrupación de módulos que no dependen entre sí.

Capa inferior Agrupación de módulos de los cuales una capa superior depende.

Capa superior Agrupación de módulos que dependen únicamente de su capa inferior.

casting Anglicismo informático que significa transformar un tipo de variable a otro.

cube Acronimo de stm32cubeIDE.

Dependencia Expresión que indica que una instancia de código no puede funcionar sin la existencia de otra.

DMA Direct Memory Access, unidades de hardware capaces de operar lógica simple directamente en la memoria, con el objetivo de liberar peso al procesador.

EHW European Hyperloop Week, la semana de la competición Hyperloop.

FD-CAN Protocolo de comunicación muy utilizado en el mundo de la industria automovilística.

FPGA Pequeño bloque de circuitos programables a través de transistores que activan o desactivan las conexiones de las entradas con puertas lógicas. Se diferencia de un microcontrolador en que la FPGA no usa una arquitectura de computador basada en un conjunto de instrucciones decodificadas por un circuito inmutable (al cual se denomina procesador o controlador, dependiendo de su arquitectura).

Generación Espontánea Programa de la UPV para apoyar proyectos impulsados por el alumnado con ayudas monetarias, didácticas y ofreciendo instalaciones.

Git Programa de gestión de versiones muy utilizado en el mundo de desarrollo de software.

H7 Familia de microcontroladores de la serie STM32 sobre la que se ha diseñado y probado la librería ST-LIB.

HAL *Hardware Abstraction Layer*; librería para programar microcontroladores proveída por STMicroelectronics.

HALAL HAL Abstraction Layer; sinónimo de ST-LIB Core.

hard-code Escribir el código directamente con la menor cantidad de referencias a valores o variables posibles, incrustando los datos directamente en las funciones.

I2C *Inter Integrated Circuit* Protocolo de comunicación basado en el paradigma Controlador - Trabajador **IDE** Anglicismo de Entorno de desarrollo integrado.

merge Anglicismo de fusión, usado principalmente para hablar en términos de Git.

Microcontrolador Circuito integrado programable.

Nucleo Pequeña placa comercial con un microcontrolador embebido proveída por STMicroelectronics.

OrangePi Línea de microprocesadores montados en una pequeña placa portátil.

Paradigma controlador - trabajador Estructura de sistemas en la que se separan en dos o más unidades un distribuidor de tareas y uno o más trabajadores que las procesan y avisan al distribuidor de los resultados.

polling Forma de comunicación que recurre a la continua petición de datos y a la espera activa, interrumpiendo el programa hasta terminar la comunicación.

pull-request Petición dentro de Github para fusionar una rama con la rama principal

RTOS *Real Time Operative System*, o sistema operativo en tiempo real.

scrum Metodología de trabajo basada en jornadas intensivas (*sprints*) para alcanzar pequeños objetivos que dividen los requisitos del proyecto.

SPI *Serial Peripheral Interface*; Protocolo de comunicación basado en el paradigma controlador - trabajador.

STM32 serie de microcontroladores de 32 bits fabricados por STMicroelectronics.

stm32cubeIDE IDE de programación de microcontroladores creado por STMicroelectronics.

STMicroelectronics empresa francesa fabricante de microprocesadores y microcontroladores programables.

ST-LIB Conjunción de todas las capas que forman la librería.

ST-LIB Core Capa base de la ST-LIB.

ST-LIB Low Capa intermedia de la ST-LIB.

ST-LIB High Capa superior de la ST-LIB.

Timer Contador de reloj en inglés. Dentro del documento se usará este término para referirse al módulo que agrega esta función mientras se usa el término en español para referirse a estos contadores de hardware.

Variables ambiente Variables que utiliza el compilador de C/C++ para tomar decisio-

nes.

2. Formato del documento

Los números rojos en texto (2) hacen referencia a figuras dentro del documento, y están vinculados a esta.


Los números rojos como superíndice¹; sin embargo, indican notas a pie de página

Los números verdes [1] hacen referencia a un enlace externo en la bibliografía al final del documento, y están vinculados a esta (y no al enlace externo)

El texto en itálica hace referencia, generalmente, a anglicismos como *array*; o a ejemplos de una composición de código fuera de un fragmento como *git clone enlace*

El texto en negrita hace referencia a nombres de variables dentro del código, como **pointer**.

Los fragmentos de código de múltiples líneas están insertados en un bloque negro, generalmente dentro de una figura:



```
println("Hola Mundo")
```

Diagrama 1: Figura con ejemplo de código

¹Nota a pie de página de ejemplo

3. Objetivo del proyecto

El propósito principal de la librería es ayudar al desarrollo del vehículo del equipo Hyperloop UPV para la competición de EHW (European Hyperloop Week). Para entender apropiadamente las razones por la que la desarrollamos y las decisiones que tomamos, primero habrá que explicar de forma resumida que es la EHW. [2]

3.1. EHW

La EHW es una competición internacional organizada por múltiples universidades europeas para promover el desarrollo de nuevas tecnologías, así como animar a los estudiantes universitarios a meterse en el mundo de la investigación y a ponerse retos cada vez mayores.

La competición se divide en dos partes. La primera es la fase de desarrollo, que cubre prácticamente todo el año escolar. Esta parte se basa en la entrega de múltiples documentos técnicos que describen el vehículo, así como las nuevas tecnologías desarrolladas para este. La otra parte es la presentación, que sucede a mediados de Julio, durante una semana que da nombre a la competición. Aquí es cuando el trabajo de todo un año da su fruto, y se muestra el vehículo en funcionamiento y se le somete a varias pruebas para mostrar estas nuevas tecnologías que fueron documentadas en la primera parte. Al terminar la EHW queda recoger y comenzar un nuevo ciclo.

Lo especial de esta competición es que la fase de desarrollo dura solamente un año. Además, aunque no sea obligatorio presentar un nuevo vehículo cada año, como el objetivo principal de la competición es la mejora y la innovación, es más que recomendable. Al tener solo un año para preparar un vehículo funcional y que cumpla los requisitos de seguridad de la EHW, el tiempo es un recurso extremadamente importante.

Debido a esto, el equipo Hyperloop UPV se ha propuesto crear un código reusable, simple de usar, testeado y flexible. Este código es la librería ST-LIB. Su objetivo es, por tanto, ahorrar la mayor cantidad de tiempo al desarrollo de código para la electrónica del vehículo, al mismo tiempo que ofrece mayor seguridad al tener un código probado y perfilado decenas de veces.

La simplicidad es necesaria para que los nuevos miembros no tengan que sufrir una curva de aprendizaje demasiado empinada, que perjudique su instrucción en la materia o incluso los lleve a abandonar el proyecto al ser abrumados por su dificultad. La reusabilidad servirá para evitar desarrollar una librería nueva cada año, y se pueda dedicar esos recursos en su lugar a expandirla y mejorarla. Por último, se necesita que el código esté testeado y sea flexible para evitar encontrar fallos o necesidades no cubiertas en un futuro al trabajar con la librería.

3.2. Uso externo

Inicialmente, la ST-LIB estaba diseñada en torno a las necesidades del proyecto Hyper-loop H8, y no se tenía en consideración prepararla para uso público. No obstante, a lo largo de la fase de diseño y desarrollo se fue viendo esta posibilidad cada vez más factible, debido principalmente a las necesidades de diseño impuestas y a la versatilidad de la HAL; una macro librería de abstracción de la programación ensamblador para arquitecturas stm32 la cual funciona para todas las familias.

Así pues, se decidió hacer pública la ST-LIB a final de la competición. El público objetivo es principalmente equipos interesados en la metodología de desarrollo scrum con objetivo de sacar un producto funcional rápidamente. Grandes empresas con objetivos a largo plazo se encontrarán con que la ST-LIB no corre código suficientemente rápido en comparación a hard-codear ², lo cual requerirá procesadores más veloces, que desemboca en mayor coste de producción. El usuario potencial de esta librería son equipos pequeños o bien proyectos sin una cadena de producción; como lo son mocks funcionales, productos de diseño (creados para un único cliente), y herramientas de testeo para la propia empresa.

3.3. Estado del arte

Se decidió crear la ST-LIB debido a que en el mercado actual las librerías de abstracción de microcontroladores está orientado exclusivamente a proyectos a largo plazo de producción masiva. La razón de esto es la existencia de los microprocesadores, unidades de mayor potencia y costo diseñadas para manejar RTOS completos como lo son Linux, Windows y MacOS.

Normalmente, para proyectos rápidos con holgura económica se usan estos microprocesadores que actúan como pequeños ordenadores que corren scripts en lenguajes de programación de alto nivel; y para proyectos de largo plazo con objetivo de abaratar costos de producción se utilizan los microcontroladores con código compilado a lenguaje máquina.

El problema viene cuando tratas de utilizar estos microprocesadores en un proyecto de ingeniería. Los microprocesadores se calientan, consumen mucha energía, su precio es notablemente superior, y al estar pensados para usar un sistema operativo suelen tener muchos más puntos de fallo, los cuales se desearían evitar en un sistema crítico. Adicionalmente, encontrar soporte para el tipo de proyecto en el que trabajas resulta más difícil, pues al fin y al cabo no es su mercado objetivo.

Esto deja al diseño de prototipos en ingeniería en un estado bastante abandonado, en cuanto a firmware se refiere. Se requieren a unas personas muy especializadas tanto en el sector de la electrónica como de la programación para que puedan sacar proyectos con una velocidad razonable trabajando en estos microcontroladores, pues muchas veces instalar los microprocesadores dentro del producto es simplemente inviable.

²Hacer código con el menor nivel de abstracción posible, poniendo la optimización por encima de cualquier otra propiedad del código

Muchas empresas recurren a crear una pequeña librería que aúne las macros más comunes que suelen usar en su sector, por lo que tras una década trabajando en los mismos tipos de proyecto suelen tener la competencia, los recursos humanos, y las herramientas como para poder defenderse dentro de este campo que es la creación de prototipos.

Sin embargo, la mayoría de veces estas empresas mantienen su código privado, está demasiado especializado, y suele requerir una empinada curva de aprendizaje; pues es más un subproducto de una necesidad que no un proyecto en sí mismo.

En el mercado hay varias opciones para programar en microcontroladores, e incluso varias librerías distintas para abstraer el mismo microcontrolador. Las más famosas para stm32 son la HAL y la librería LL. De las dos, la primera es la más abstracta, mientras que LL es de muy bajo nivel y está diseñada para programadores de hardware especializados. Otras empresas como MIPS, Intel, o Texas Instruments también diseñan microcontroladores que poseen sus propias librerías, pero estas son aún más especializadas y requieren de expertos en su propio campo.

Opciones más extremas simplemente acrecientan los problemas. Usar *FPGA* o programar el código directamente con circuitería consume una cantidad inmensa de tiempo; inaceptable para el propósito de un prototipo. Tratar de controlar todo con una API externa o implementar un ordenador completo y funcional a bordo suele ser demasiado aparatoso y no llegan a cumplir los requisitos de funcionalidad y seguridad que un proyecto de ingeniería suele requerir.

Al final, la ST-LIB no es más que una iteración de las librerías que las empresas que abarcan los sectores de la ingeniería suelen estructurarse; solo que esta vez sí es su propio proyecto con el propósito de que pueda ser usado a largo plazo y con facilidad por el público, y quizás ayudar a nuevos proyectos y empresas que no gozan de una librería propia a poder lanzar sus proyectos de corto y medio plazo con más facilidad.

4. instalación y uso

En este apartado se explicará como instalar los programas necesarios para usar la librería, como utilizarla, que se puede configurar e incluso como modificarla por si fuese necesario adaptarla para otros microcontroladores de stm32 o se quisiera en algún futuro ampliarla. Este apartado no explica nada de la librería que no se puedan encontrar en otros apartados y su único propósito es facilitar la instalación y uso de esta. Si no se tiene interés en el uso directo de la librería, este apartado es opcional.

4.1. Git

Para poder instalar la librería hay dos opciones. La primera es ir al Github de la ST-LIB y descargar una versión de entre las disponibles directamente, lo cual dará un comprimido portable que puede descomprimir directamente en su workspace. La segunda es usar git para copiar el repositorio. Mientras que ambas opciones son válidas, es recomendable utilizar git para facilitar el manejo de versiones, ya que hará más cómodo cambiar entre estas e incluso crear un repositorio aparte para guardar la librería modificada con su propio historial de versiones.

La instalación de Git es muy sencilla[3] y esta claramente explicada en su página web[4]. La única complicación que se puede dar es que en el caso de Windows no se añada al path correctamente, dependiendo del método de instalación elegido. Si se da el caso, en el buscador de windows escriba “Path”, abra “Editar las variables de entorno del sistema”, vaya a la pestaña “Opciones avanzadas”, seleccione “variables de entorno”, elija “PATH”, luego presione “Editar”, se abrirá una nueva ventana con varios botones, incluido el botón “Nuevo”. Se aprieta este botón, y se escribe la dirección en la que se instaló Git. Si no se modificó el lugar de instalación, debería encontrarse en “C:\Program Files\Git”.

Una vez se tiene instalado Git en el sistema operativo se puede usar la consola del susodicho para copiar el repositorio. Simplemente se debe escribir en la consola `cd “path”` (donde path es la dirección donde se quiere copiar la librería, recomendable que se escoja el workspace de su editor de código), se presiona Enter, asegurandose de que se haya cambiado la dirección en la que se encuentra a “path”, y por último se usa **git clone <https://github.com/HyperloopUPV-H8/ST-LIB.git>** [1] para copiar desde el repositorio remoto todo su contenido. Se recomienda también clonar el proyecto base, el cual ya tiene todas las configuraciones implementadas para correr la librería. Para esta use git clone <https://github.com/HyperloopUPV-H8/template-project.git> [5].

Una vez se tenga instalado el repositorio para cambiar de versiones, simplemente se usa el comando `git checkout versión` para cambiar la versión de la librería. Una vez todo funcione, es libre de seguir utilizando git o trabajar en archivos locales. Se recomienda que copie la instalación fuera de las carpetas git y trabaje en la copia para evitar eliminar todo su progreso en el código al cambiar de versión (git tiene una protección contra esto pero no te permitira cambiar la versión mientras esta protección esté levantada)

4.2. IDE

La librería funciona tanto en Linux, como en Windows, y como en MacOS. En esta explicación nos centraremos en la instalación en Linux (más recomendable) pero la instalación es prácticamente igual para cualquier sistema operativo.

Lo primero que se debe hacer es instalar el programa de STMicroelectronics para la programación de microcontroladores. Este programa se llama stm32cubeIDE, y está basado en eclipse, un programa de edición de código alternativo a Visual Studio especializado en programación de Java y C/C++. Al estar basado en eclipse, siempre que se quiera hacer algo en cube que no sea estrictamente de los microcontroladores (como por ejemplo gestionar vinculación de librerías) se puede buscar documentación de eclipse directamente, la cual es mucho más abundante que la del cube.

Para instalar el cube debemos ir a la página oficial de stm32 [6] y descargar un instalador para nuestro sistema operativo. Para Linux hay 3, uno para ubuntu (*deb*), uno para fedora (*rpm*) y uno genérico. La versión en la que se trabajó la librería fue la 1.10.1, así que esta es la versión preferente.

Para ejecutar el instalador en nuestro sistema Linux debemos descomprimir el .zip con unzip, y luego deberemos ejecutar el .sh en la consola dentro de la carpeta donde se haya descomprimido, con el comando **sudo sh ./st-stm32cubeide_*tu versión*.sh**. Se deberán aceptar las licencias de usuario y indicarle el directorio de instalación. Para Windows y MacOS en lugar de un comando por consola se puede simplemente ejecutar haciendo doble clic al archivo extraído.

Una vez instalado el cube se deberá añadir la librería para poder usarla. Simplemente seleccione la opción de “*Open projects from file system*”, introduzca el directorio donde se descargó (o se copió desde git) a través del botón Directory... y presione **Finish** como se muestra en el diagrama 2

Aparecerá la opción de crear un proyecto nuevo desde cero, que te dejará seleccionar tu microcontrolador y construirá un .ioc con la configuración estándar para este, el cual se tendrá que modificar a gusto del usuario. Aunque crear el proyecto desde cero y luego importar la librería sea posible llevará bastante tiempo debido a que habrá que configurar los periféricos en el .ioc y crear un archivo runes de configuración de la librería, así que para el caso de los microcontroladores H7 existe ya un proyecto configurado que se puede importar usando git y la opción de abrir un proyecto desde un sistema de archivos.

Este proyecto se denomina template-project, y como se menciona en el apartado anterior, se puede descargar en su respectivo github [5]. Lo que contiene este proyecto es un .ioc configurado para funcionar con la ST-LIB, junto a módulos de la HAL modificados y un archivo runes que tiene múltiples periféricos de ejemplo listos para utilizar.

El mayor defecto de la librería es que requiere de modificar los métodos de inicialización de los archivos generados automáticamente por el cube para que añada todos los periféricos como los desea el usuario lo cual aumenta mucho la complejidad de uso y puede lastrar el

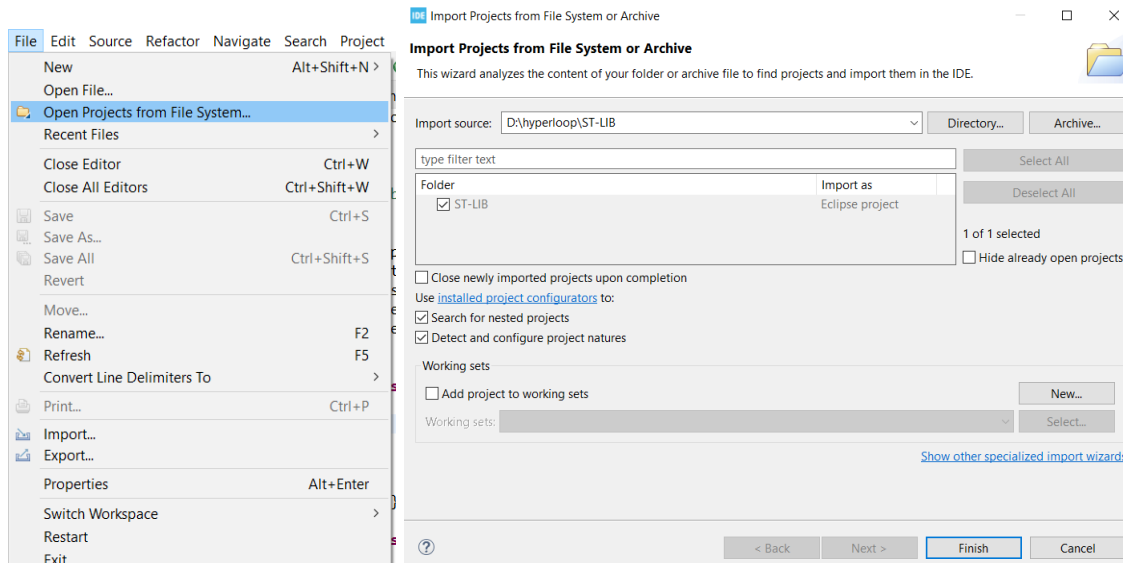


Diagrama 2: Imágenes de como importar la ST-LIB en el cube

proceso de creación de código. Por ello se ha creado el template project con el propósito de ofrecer una versión utilizable directamente nada más se descarga, pero esta viene con sus propios problemas. Principalmente, la configuración de pines con la que viene es estática, así que si se usa otra familia o se quiere cambiar la función de un pin por cualquier razón, se debe modificar todos estos archivos obligatoriamente.

Se ha intentado ofrecer todas las utilidades posibles en la configuración del template-project, pero más adelante se verá un apartado de como modificarlo para cambiar configuraciones comunes.

Ahora se deberá importar el template-project en el cube. Lo primero que se debe hacer si se le cambia el nombre al template-project es modificar el .project que se encuentra dentro de la carpeta del proyecto. Dentro de este archivo hay una línea que pone **<name>**, y ahí se deberá poner el nuevo nombre del proyecto exactamente igual. Si no se cambia, el cube simplemente no lo importará cuando se apriete al botón de importar.

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
  <name>nuevo nombre</name>
  <comment></comment>
  <projects>
    <project>ST-LIB</project>
  </projects>
```

Además de el .project, la carpeta en la que se encuentra y el .ioc deben tener este nombre, y no se puede tener importado en el cube otro proyecto que se llame igual (o que

tenga su .ioc o su <name>dentro del project igual). En cualquiera de estas situaciones, el cube no importará nada y no lanzará ningún error. Una vez todo se haya modificado para que coincida, se selecciona de nuevo la opción de “*Open projects from file system*” y se importa de la misma forma que la ST-LIB. Si al importarlo no aparece en el workspace inmediatamente lo más probable es que haya un error de los mencionados previamente.

Una vez tanto la ST-LIB como el proyecto están importados correctamente al workspace del cube queda tocar una última configuración antes de comenzar a programar. La ST-LIB está pensada para funcionar tanto en las nucleos comerciales como en placas diseñadas con un cristal de cuarzo externo. Esta configuración se ha de introducir al programa en tiempo de compilación, y como el cube usa su compilador interno para hacer esta tarea, debe indicársele en el propio proyecto las variables ambiente que se usan.

Poniendo el cursor encima del proyecto ST-LIB importado, se pulsa el botón derecho del ratón, se selecciona “*properties*”, y dentro de C/C++ general >Path and Symbols >Symbols se modifican tanto GNU C como GNU C++. En el primero se pone en HSE la frecuencia del reloj global del microcontrolador deseado (para las nucleos de la familia H7 es 80000000MHz, y se escribe como 80000000UL) y se dejan los símbolos tanto de NUCLEO como de BOARD sin ningún valor asociado. En GNU C++ se deja solo el símbolo que represente la configuración deseada, NUCLEO para usar el cristal de cuarzo interno y BOARD para usar un cristal externo (relojes HSI y HSE respectivamente).

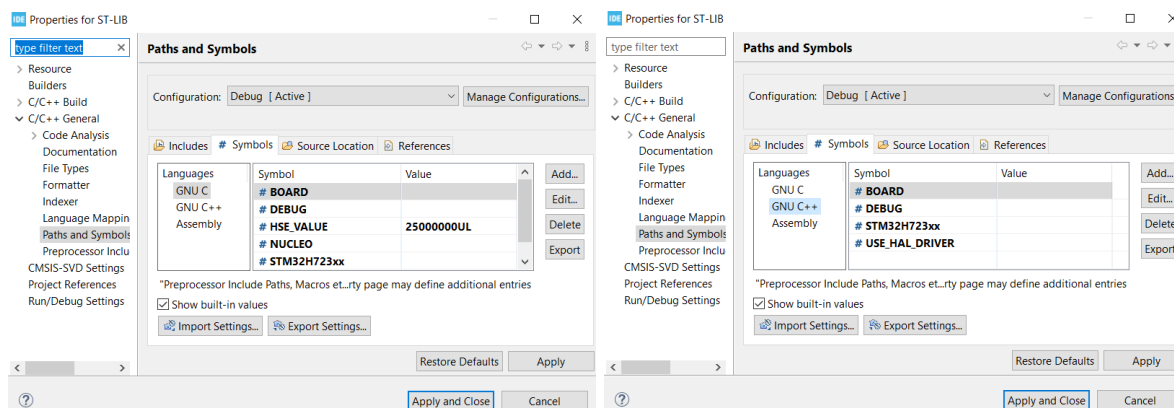


Diagrama 3: Ejemplo de configuración para una placa con cristal externo a máxima velocidad de un microcontrolador H7

Luego se aplican las mismas modificaciones en el proyecto y ya estaría listo. Solo quedaría probar si los relojes están configurados correctamente, la forma más fácil de hacer esto es hacer que un LED conocido parpadee cada segundo usando el módulo time y comprobar que efectivamente no parpadea ni demasiado lento ni demasiado rápido.

4.3. Comenzando a usar la librería

Una vez se tiene instalada la librería y se ha descargado el project template (o alternativamente se ha hecho un proyecto propio que importe la librería), toca configurar la librería para poder utilizarla dependiendo de las necesidades de cada usuario.

Lo primero que se ha de hacer para usar la librería si no se va a usar los módulos de comunicación Ethernet es desactivarlos. Estos módulos tienen grandes utilidades; pero ocupa varios pines, puede ralentizar la ejecución de la placa en el orden de microsegundos, y puede ser bastante aparatoso.

El caso es que el módulo Ethernet se encarga por si mismo de gestionar y proteger la comunicación, y una de las protecciones es **no activar la placa hasta que el bus Ethernet se establezca correctamente**. Por ello, si no se va a usar, se debe o comentar dentro de los métodos HALAL::start() y ST-LIB::update() todas sus apariciones (dos en start + una en update); o alternativamente desactivar el módulo como se mostrará en el próximo sub-apartado.

4.3.1. Activar y desactivar módulos

Los módulos se activan y desactivan a partir de los *define* de la HAL, la librería sobre la que la ST-LIB se basa. Con ello se evita que esté funcionando el módulo en la HAL sin estar activo en la ST-LIB y viceversa.

Para desactivarlos se debe ir dentro del proyecto a los archivos *include*³ a un archivo llamado stm32h7xx_hal_conf.h⁴ y simplemente comentar o descomentar los *define MODULE_ENABLED* del módulo que se desee activar o desactivar.

Por ejemplo, el módulo Ethernet de la HAL *HAL_ETH_MODULE_ENABLED* se encuentra dentro del project_template en la línea 48 de stm32h7xx_hal_conf.h, y viene descomentado de base. Este es el más común que se va a querer activar o desactivar; pues es una gran herramienta pero puede impedir el correcto funcionamiento de la placa si no se está utilizando. Simplemente se ponen dos barras justo detrás suya para comentarlo y ya está desactivado.

4.3.2. Configurar Pins y Runes

Al usar la ST-LIB puede requerirse reconfigurar los pines si las opciones ofrecidas de base por el project template son insuficientes. Además, si se hace un proyecto desde cero, se necesitará configurar un archivo Pins.hpp y un archivo Runes.hpp para que la ST-LIB funcione correctamente.

³la carpeta de archivos que en la mayoría de proyectos C / C++ contiene los header files

⁴Si no se está usando la familia H7, este archivo tendrá en el nombre la familia que se este usando en su lugar; ej. stm32f3xx_hal_conf.h para la familia F3

Configurar Pins es bastante sencillo, se crea un objeto Pin que indique el puerto y pin adecuado (siendo el puerto la letra y el pin el número); y si se requiere darle una funcionalidad alternativa se indica además en su constructor esta funcionalidad alternativa.

La funcionalidad alternativa es una configuración que se le da a los pines cuando se quiere utilizar utilidades especiales que no entren en el uso de periféricos básicos (INPUT, OUTPUT, ANALOG, EXTI, o NOT_USED).

Lo que pueda hacer cada Pin con estas funcionalidades depende de la placa que se esté usando, y por tanto requiere de revisar la ficha técnica del microcontrolador que se esté usando. Por ejemplo, no todos los pines pueden hacer de PWM. El project template ya viene con una configuración ejemplo en el que se da todas las opciones para la familia H7, pero si se quiere usar otra familia u otra configuración se debe revisar la ficha técnica para poder usar las funciones alternativas.

Como un pequeño indicador para la familia H7; AF0 y AF15 son control de sistema, AF11 son Ethernet, AF14 es UART, y el resto son relojes (principalmente para PWM) y otros protocolos de comunicación. A continuación, un pequeño extracto de project template para que se vea como se declaran los pines:

```
Pin PE2(PORT_E, PIN_2);  
Pin PE3(PORT_E, PIN_3);  
Pin PE4(PORT_E, PIN_4, AF4);  
Pin PE5(PORT_E, PIN_5, AF4);  
Pin PE6(PORT_E, PIN_6, AF4);  
Pin PC13(PORT_C, PIN_13);  
Pin PFO(PORT_F, PIN_0, AF13);  
Pin PF1(PORT_F, PIN_1, AF13);  
Pin PF2(PORT_F, PIN_2, AF13);  
Pin PF3(PORT_F, PIN_3, AF13);
```

Diagrama 4: Ejemplo de configuración de Pins.hpp extraído de project template

En cuanto al Runes.hpp, ahí se deberán declarar tanto los **handlers** de la HAL necesarios como las instancias de los periféricos que luego usarán los métodos **inscribe** de cada módulo. Lo mejor para hacer esto es comenzar desde arriba e ir hacia abajo.

Como ejemplo, se quiere crear una nueva instancia de UART que utilice los pines PB6 y PB7 para comunicarse. Para ello primero se ha de mirar que valores contiene la estructura de instancia UART. Esto se puede ver o bien con el asistente de código de él cube (presionando Ctrl + espacio), o abriendo el header de UART. [5](#)

```
struct Instance{
    Pin TX; /**< Clock pin. */
    Pin RX; /**< MOSI pin. */
    UART_HandleTypeDef* huart; /**< HAL UART struct. */
    USART_TypeDef* instance;
    uint32_t baud_rate;
    uint32_t word_length;
    bool receive_ready = false; /**< Receive value is ready to use pin. */
    bool initialized = false;
};
```

Diagrama 5: Estructura de instancia de UART

Dentro del header del UART se puede ver que requiere ambos pines que se van a usar (PB6 y PB7, estos se deberán declarar con la función alternativa UART como se explicó anteriormente en este apartado), un **handler** de la HAL; una instancia de la HAL, una variable denominada `baud_rate` (que define la velocidad a la que queremos comunicarnos, en bits por segundo) y otra variable `word_length` (que define el tamaño de palabra, se recomienda usar los *enum* de `UART_WORDLENGTH` pues definen todos los valores posibles para este protocolo)

Para los **handlerTypeDef** basta con declararlos sin definirlos al principio del documento `Runes.hpp`

```
#pragma once
#include "Pins.hpp"

UART_HandleTypeDef huart1;
```

Diagrama 6: Ejemplo de donde podría ir un handler

En cuanto a los **TypeDef** normales suelen estar ya declarados con el mismo nombre más un número. Lo mejor es usar los que ya vienen con la librería HAL pues existen tantos como la placa que se este usando puede configurar. Asumiendo que se desea un baud rate de 9600 y un tamaño de palabra de 64 bits la configuración sería como en la figura 7

```
UART::Instance UART::instance5 = { .TX = PB6, .RX = PB7, .huart = &huart1,
    .instance = USART1, .baud_rate = 9600, .word_length = UART_WORDLENGTH_64B,
};
```

Diagrama 7: Ejemplo de declarar en el `Runes.hpp` una nueva instancia

Cual **handler** y cual instancia de UART escoger depende de como estén conectados los pines en el microcontrolador, así que será necesario mirar en la ficha técnica para los pines específicos a que UART están conectados. Alternativamente se puede abrir el .ioc para ver qué opciones tiene el pin; pero se recomienda no modificarlo o si no este generará código nuevo pisando el código que se tuviese anteriormente.

Si los pines no están conectados al mismo UART o uno de los dos no está conectado a ninguno, no se podrán utilizar (es una limitación de hardware).

Una vez se tiene la instancia creada, solo queda mapearla a un periférico disponible en la clase UART y ya se podría utilizar. Para hacer esto, se ha de definir el mapa de UARTs disponibles *available_uarts*. En caso de que ya este definido en otro sitio, simplemente se debe añadir a esa definición la nueva conexión en el mapa, de esta manera:

```
unordered_map<UART::Peripheral, UART::Instance*> UART::available_uarts = {  
    {UART::uart5, &UART::instance5},  
};
```

Diagrama 8: Ejemplo de como vincular en el mapa el periférico y su instancia

```
UART_HandleTypeDef huart1;  
  
UART::Instance UART::instance5 = { .TX = PB6, .RX = PB7, .huart = &huart1,  
    .instance = USART1, .baud_rate = 9600, .word_length = UART_WORDLENGTH_64B,  
};  
  
unordered_map<UART::Peripheral, UART::Instance*> UART::available_uarts = {  
    {UART::uart5, &UART::instance5},  
};
```

Diagrama 9: Ejemplo completo de declaración del UART en Runes

Una vez hecho esto, ya se podría; al fin, utilizar los pines PB6 y PB7 como UART; asumiendo que se hayan configurado correctamente en Pins.hpp como se indicó anteriormente. Para ello simplemente se hace **UART::inscribe(uart5);** justo antes del **ST-LIB::start()** (o **HALAL::start()**, dependiendo de cuanto de la librería se quiera activar) y se utiliza como se muestra en su sub-apartado.

Todos los módulos de la librería usan la misma estructura de instancia y mapa; aunque algunos puedan usar más de un mapa. Si se sigue este proceso se puede configurar cualquier Pin a cualquier módulo sin ningún problema; pero si se tiene alguna duda siempre se puede

mirar el Runes.hpp del template-project para usarlo como ejemplo, pues tiene todos los módulos configurados en al menos una instancia.

Esta es la configuración más complicada de la librería, y una transacción necesaria si se quiere usar el microprocesador a su máximo potencial. Sin embargo, para la mayoría de prototipos y programas básicos, el ejemplo de template-project debería tener todas (o al menos la mayoría) de las necesidades cubiertas, por lo que se recomienda utilizarlo y solo modificar el Runes si es estrictamente necesario para el diseño que se tiene en mente, pues puede consumir bastante tiempo y se pueden encontrar problemas de configuración de la HAL.

4.3.3. Prueba básica de funcionamiento

Una vez toda la configuración está en marcha, toca probar si está todo funcionando correctamente. Para ello, se puede usar un pequeño trozo de código que use la ST-LIB para hacer parpadear un led, y comprobar si la librería está funcionando.

```
#include "main.h"
#include "lwip.h"

#include "ST-LIB.hpp"
#include "Runes/Runes.hpp"

int main(void){
    DigitalOutput led(PE1);
    ST-LIB::start();
    time::add_low_precision_alarm(1000,()[]{led.toggle();});
    while(1){
        /*Este bucle evita que termine el programa*/
        /*normalmente aquí dentro se pondrían acciones secundarias del programa*/
    }
}
```

Diagrama 10: Ejemplo de archivo main.cpp de un led parpadeando una vez por segundo

Este programa declara un led⁵ y se añade una función a una alarma cada segundo que cambie el estado del led.

Ahora se revisarán los posibles problemas que pueda tener el programa.

Si el led no parpadea lo primero que hay que hacer es comprobar en el *debugger* del cube si pasa del **ST-LIB::start()**. Si no pasa del start lo más probable es que no se haya desactivado el módulo Ethernet ni se tenga un bus Ethernet viable conectado (viable quiere decir que esté conectado en ambos puntos a un dispositivo con capacidades de resolver ARP request, ósea se, que pueda usar Ethernet). Si sobrepasa el **start** y activa

⁵en este caso se ha puesto en el PE1, pues es el led amarillo configurado en el template project para la nucleo stm32h7z23zg, pero se puede configurar para cualquier pin que no tenga una función alternativa; pues es un GPIO

el **led.toggle()** quiere decir que el Pin está configurado en el Runes.hpp o en Pins.hpp como alguna función alternativa.

Si el led parpadea, pero no cambia de estado cada segundo exactamente, entonces se configuraron mal los relojes en el apartado del IDE. Quizás no se pusiera la variable de entorno **nucleo** o **board** correctamente, o quizás el reloj externo que se usa (en caso de seleccionar board) no tenga la frecuencia en Hz que se escribió en la variable de entorno HSE.

En caso de que parpadeé correctamente cada segundo, la librería debería funcionar correctamente y estar lista para ser usada.

5. Desarrollo de la librería

En esta sección se verá una memoria de como se desarrolló la librería, explicando las condiciones sobre las que se trabajaba, los objetivos principales de la librería.

5.1. Métodos de gestión de proyecto

Para estructurar un proyecto en grupo se necesita una organización clara. La jerarquía del equipo usada fue una estructura organizativa funcional de poca verticalidad. Se requiere de una estructura organizativa funcional debido a que el proyecto Hyperloop es multitudinario, y los sub-sistemas deben saber a quién dirigirse a partir de su competencia cuando requieren a personas de campos específicos. El proyecto de la ST-LIB entraba dentro del subsistema de firmware, y requería de comunicación directa y continua con el equipo de software y hardware para conocer las especificaciones funcionales necesarias que debía cubrir la librería.

Esta estructura se dividió en tres capas, los miembros y colaboradores del equipo, los PM (*project manager*) que gestionaban un subsistema entero, y los capitanes que dirigían a todo el equipo a partir de comunicarse principalmente con los PM y para casos específicos directamente con los miembros y colaboradores más instruidos en la tarea. Aunque hubiese una jerarquía de equipo se dejaba abierta la comunicación directa entre todo miembro del equipo, y se usaba la responsabilidad y confianza en los miembros como único muro para no abrumar a los PM y capitanes con dudas y consultas varias. Este tipo de método de trabajo, si se consigue llevar a cabo, es muy ágil y flexible, pero es más vulnerable a fallos individuales y puede llegar a sobrecargar a miembros del equipo clave si no se trata con cautela. Se requiere de personas capaces de trabajar en equipo, así que la instrucción para este tipo de metodologías es obligatoria para cada nuevo miembro.

Para el subsistema de firmware se usó específicamente una metodología scrum ágil basada en fases de proyecto con sprints y hiatos de planificación para las consecuentes fases y para tratar posibles retrasos. En cada sprint se le asignaba a cada miembro unos trabajos que se debían llevar a cabo en un marco de tiempo, y al final de este se revisaba cuantos de estos objetivos se habían cumplido y se trataban los posibles retrasos así como se aprovechaban los posibles adelantos redistribuyendo trabajo.

Esta metodología pone responsabilidad en cada miembro al tener una forma objetiva de marcar su trabajo. Lo bueno de esto es que permite aligerar la carga de trabajo de los PM y capitanes que debido a la estructura de equipo corren riesgo de sobrecarga. Lo malo es que al ofrecer una parte del trabajo a cada miembro y planificar teniendo en cuenta lo que van a completar un miembro disfuncional, ausente, o que sufra un contratiempo puede provocar un fuerte cuello de botella y dañar el ritmo del equipo entero.

Para mitigar lo máximo posible la rotura del ritmo de trabajo, los sprints se reducen a una semana de tamaño, haciendo reuniones a mediados y final de sprint para redistribuir

el trabajo y analizar los avances, permitiendo rectificar problemas en el avance del trabajo en pocos días, siempre y cuando los miembros sean comunicativos.

En cuanto a las fases del proyecto, que definen hitos importantes del avance del proyecto, se dividen en fase de diseño, fase de testing y fase de presentación. La fase de diseño incluye la construcción del equipo, la comunicación de requisitos, el diseño de la estructura de la librería y la implementación mínima funcional de esta, en ese orden. Como esta fase de diseño sucede a la vez que la fase de diseño de otros subsistemas, es común la retroalimentación de requisitos y la reestructuración de la librería mientras se está en fases más avanzadas, por problemas que se puedan encontrar o mejoras posibles que se puedan añadir. Ahí es donde brilla la naturaleza iterativa del scrum, permitiendo añadir a los sprints cambios de la estructura y comunicarlos rápidamente al equipo sin afectar al ritmo de trabajo.

Una vez se termina la fase de diseño se hace un *release* de la librería (más adelante se elaborará en el concepto de *release*) con una funcionalidad probada mínima y puede comenzar la fase de testing. Esta fase es la más intensa debido a que es aquí donde se encuentran la mayoría de errores y contratiempos. Se basa en utilizar la librería para implementar múltiples placas con requisitos funcionales críticos usando todas las herramientas de la librería y hacer pasar a estas placas unos tests tanto de seguridad como de capacidades funcionales.

Es posible que en esta fase se requieran cambios en el diseño de la estructura de la librería como el añadido de nuevos módulos o mejorar las capacidades de módulos ya existentes, lo cual puede afectar a su vez a otros módulos de la librería provocando un efecto cascada de trabajos urgentes y fallos en la librería. Por ello en esta fase se requiere de horarios flexibles, una comunicación muy abierta, y de trabajo preventivo. En esta fase se hacen múltiples releases que arreglan errores, añaden funcionalidades y confirman revisiones al código. Al terminar esta fase la librería debe estar en su versión final, debe cumplir todos los requisitos y estar lista para actuar en sistemas críticos.

La última fase es la fase de presentación, que dura alrededor de un mes. Aquí se junta el trabajo de todos los subsistemas, se termina el producto, se planifica las formas de actuar a la hora de la presentación y finalmente se presenta en la EHW. Esta fase sería la equivalente a los meses después de la salida al mercado de un producto, donde el equipo recibe feedback y hace unos cambios mínimos de última hora para arreglar fallos que no se pudieron ver sin tener el producto final. Si por alguna razón se requiriesen cambios drásticos en la librería, querría decir que la metodología de trabajo ha fallado.

5.2. Gitflow

El Gitflow es una rama de la metodología de gestión de proyectos orientada principalmente al control de archivos y documentos del proyecto, pero como esta parte es tan esencial en un proyecto de desarrollo de software como lo es la creación de una librería,

se le ha dado un apartado propio para evitar saturar la sección de métodos de gestión de proyectos.

Gitflow es una estructura y planificación del uso de programas de gestión de versiones (principalmente Git, del cual viene su nombre) que define los pasos a seguir para añadir cambios o introducir nuevos archivos al espacio de trabajo del equipo sin interferir en el trabajo de otros. El Gitflow se basa en el uso de ramas (o “*branches*”) de trabajo que se encargan de controlar partes específicas de los documentos (en nuestro caso, los módulos de la librería) que hacen una instantánea del estado actual del espacio de trabajo y te deja modificar esta instantánea sin afectar al espacio en común del equipo, y por ende evitando daño directo al avance de los compañeros. Una vez se ha terminado de trabajar en la rama, se debe reunir los avances hechos con el espacio de trabajo del equipo, en un acto conocido como fusión de las ramas (o “*merge*”)

En nuestro espacio de trabajo en común está compuesto por dos ramas, **development** que guarda todos los avances que fueron fusionados, y **main** que guarda la última instantánea del proyecto que ha sido correctamente testeada y su funcionalidad está asegurada.

Para poder juntar los avances hechos por un miembro del equipo con **development** (la rama de trabajo en común) se debe tener un mínimo de control para asegurar que no se ha cometido un fallo que comprometa la funcionalidad de la rama **development**. Para ello se usa las *pull-request*, una funcionalidad de Git que permite a otros miembros del equipo que no hayan trabajado en esa rama revisar los cambios, pedir modificaciones a la rama a fusionar, y finalmente aceptar los cambios. En nuestro caso en concreto, se requería que la mitad del equipo de firmware (2 personas + el propietario de la rama) aceptara la *pull-request* de la rama.

Para hacer una nueva *release* (fusionar *development* con *main*) se reúne al equipo entero en una reunión especial y se hace una *pull-request* que requiere de la aceptación del 70 % del equipo (todos menos 1) para completar esta.

Una vez la librería está completa comienza la fase de testeo, donde el Gitflow se modifica para facilitar lo máximo posible la salida y aprovechar las nuevas herramientas que se fueron desarrollando en paralelo a la librería. Para ello lo primero que se hace es reducir el número de personas que aceptan la *pull-request* para fusionar con **main** en 1 más la herramienta de testeo automático (un servidor desarrollado para la fase de testing que corre pruebas en el código del pull request para comprobar que el código no se rompió en los cambios de esta, funcionando 24 horas). Esto reduce mucho la dependencia entre los programadores de la librería permitiéndoles sacar mucho más rápido los fixes.

Además, se implementó un control de versiones de la rama **main**, permitiendo tener varias versiones funcionando al mismo tiempo; principalmente para el caso en el que una placa funcionase completamente con una versión pudiera congelarse sin tener que revisarse para las nuevas versiones, aunque también como seguro en caso de que un *merge* estropeara la librería de una forma imprevista, ya que el control sobre la rama **main** fue reducido a favor de mayor facilidad a la hora de revisarla.

El último cambio al Gitflow entre las fases de desarrollo y testeo fue añadir como requisito a las actualizaciones a main un log file que explicara los cambios que se han producido en la librería a nivel funcional con cada nueva actualización. Estos cambios además no debían afectar a la interfaz de la librería a menos que fuese absolutamente necesario.

En cuanto a los programas usados para aplicar el Gitflow los principales fueron Git, GitHub, Visual Studio Code, GitKraken, y el formato Markdown para la escritura de la Wiki del proyecto. En Git se utilizaban dos repositorios separados, uno para la propia librería y otro para el proyecto de las placas que servía, entre otras cosas, para probar la librería. Había una dependencia unidireccional entre la librería y el proyecto de los microcontroladores, por lo que el segundo debía actualizarse rápidamente a los cambios del primero. Para modificar este último, se usaba un Gitflow más simplificado sin *pull-requests*, permitiendo cambiar mucho más rápido el proyecto de las placas a cambio de tener el riesgo de que este fallara. Este riesgo era únicamente aceptable gracias a que la librería no dependía del susodicho, y por tanto los fallos en el proyecto de las placas no afectarían gravemente al avance del trabajo.

Respecto a GitHub, Visual Studio Code y GitKraken eran soportes de alto nivel para facilitar el uso y entendimiento de Git a los miembros del equipo menos versados en el uso de esta herramienta. Visual Studio Code permite hacer cambios rápidos y controlar conflictos de forma más visual en las ramas, y tiene atajos para las acciones más usadas dentro de Git. GitKraken permite observar de una forma más “*user friendly*” los movimientos entre ramas para ver si estaban habiendo problemas a la hora de aplicar el concepto del Gitflow en la práctica. Se podían ver de qué ramas a cuáles se habían hecho los merge y los *pull-request*, permitiendo alertar *branching* excesivo y resaltando las ramas más problemáticas. GitHub cerraba el círculo ofreciendo control sobre las *pull-request*, los comentarios, y dando la capacidad de modificar la estructura del Gitflow en minutos en caso de necesidad.

Los módulos de la librería (que en el Gitflow son tratados como ramas) se separaron en tres grandes bloques según su nivel de abstracción: Core, Low y High. Cada nivel requería de distintos conocimientos y cantidad de comunicación, además de que cada nivel superior tenía dependencias con los niveles previos. Para que esto fuese posible, se requiere que el grafo de dependencias entre módulos de la librería cumpla unas normas, debe ser un grafo multinivel, cuyos niveles sean los tres bloques. Esto quiere decir, a grosso modo, que ningún módulo de la librería puede depender de otro módulo del mismo nivel o de un nivel superior. Diseñar la librería así tiene otra gran ventaja añadida, y es que mientras se siga este concepto de diseño es imposible sufrir dependencias cíclicas, pues para que un grafo sea multinivel debe, entre otras cosas, ser acíclico.

La ST-LIB Core es el bloque que requería más conocimientos de hardware y firmware para ser creada, abstraía directamente los registros de Hardware, apoyándose únicamente en la HAL, la cual requiere de conocimientos tan amplios como trabajar directamente con

los registros. Esta es la que tendría más peso en la eficiencia del código, la seguridad, y la flexibilidad de este. La mayor ventaja es que es un punto de apoyo para las capas superiores y no está diseñada con la intención de ser usada directamente, así que podía ser tan obtusa y fea en su uso como fuese necesario para cumplir los requisitos dados por los otros subsistemas.

La ST-LIB Low es la capa intermedia entre el usuario y el corazón de la librería, y su objetivo principal es traducir a conceptos de lenguajes más abstractos las herramientas dadas por la ST-LIB Core. Pasar de uso de ids y registros a estructuras, clases y POO, gestionar los posibles fallos y recuperarse de errores o datos aberrantes, ayudar a depurar el código al usuario, y hacer macros de estimaciones y cálculos usados en múltiples capas son algunas de las funciones que tiene esta capa.

Por último, la ST-LIB High tiene como objetivo juntar todos los módulos de la ST-LIB Low en grandes macros como *“iniciar”*, *“terminar”*, *“bucle de trabajo”*, o *“interrupción”*, así como activar o desactivar los módulos que están en uso, imponer protecciones entre módulos, y facilitar la creación de macros específicas para cada placa.

6. Diseño y funcionalidad

En esta sección se explicará en más detalle la estructura de la librería, como funciona y cuáles son los casos de uso esperados para cada uno de sus módulos. Se dividirá en las tres partes principales de la librería: Core, Low y High, más una explicación generalizada del objetivo de diseño y un par de sub-apartados adicionales para estructuras de especial interés.

6.1. ST-LIB Core

La ST-LIB Core es la capa más profunda de la librería ST-LIB, y por tanto, sufre de una curva de aprendizaje más empinada y puede ser más laborioso a la hora de trabajar con ella.

Las capas superiores se han hecho con la intención de que el usuario no tenga, necesariamente, que saber utilizar la ST-LIB Core; por lo que este apartado no es necesario para comenzar a utilizar la librería directamente.

No obstante, saber utilizar esta capa puede ser una muy buena base para aprender en profundidad como funciona tanto la librería como la programación de microcontroladores en general; y es la capa que permite hacer más cosas al estar más cerca del código máquina.

Si se quiere hacer un código básico para un evento urgente (como terminar un prototipo para una fecha límite cercana), es recomendable saltarse este apartado. No obstante, si se tiene pensado usar la ST-LIB a largo plazo para múltiples proyectos, o si ya se tiene experiencia en el mundo de los microcontroladores; este es el apartado más importante.

6.1.1. Modelos y Servicios

La ST-LIB Core se divide en la parte estructural (o los modelos) y la parte funcional (o los servicios). Los modelos permiten abstraer el concepto de registros de periféricos, relojes, contadores, uso de la memoria flash, y demás propiedades del propio hardware del micro que pueden ser demasiado abrumadoras para una persona que se está introduciendo al mundo del firmware y demasiado repetitivas y laboriosas para un equipo que ya conoce sus necesidades sobre estos aspectos del hardware y prefieren tener una base ya creada para no tener que configurar cada proyecto que hagan.

Los servicios, por su parte, abstraen las macros más comunes en el mundo del firmware, como iniciar periférico, crear interrupción, atender interrupción, reconfigurar reloj, crear alarma, atender alarma, y lecturas y escrituras de todo tipo. Los servicios de la ST-LIB Core son primitivos y de un nivel de abstracción muy bajo, y solo existen como soporte para niveles de abstracción superiores y como opción a configurar para expertos de firmware. Las funciones que hacen son muy simples, pero reducen la necesidad de 9 líneas de código a 1, y gestionan los errores más comunes cuando se hacen proyectos a gran escala con múltiples micros, como lo son no haber declarado o iniciado un Pin, equivocarse en un

número y llegar a valores peligrosos para el micro, o usar un Pin configurado de una forma para una función que requiere otra configuración.

La mayoría de veces gestiona estos errores a través del ErrorHandler, una clase que se dedica específicamente a dar información retro-alimentada al programador a partir de mensajes en el protocolo de comunicación preferido (Viene configurado para usar el protocolo FD-CAN, pero con un cambio a dos líneas de un método se pueden usar cualquier otro de los protocolos ofrecidos por la librería), y de parar el micro en caso de fallo de una forma más segura en lugar de usar un *Hard Fault*.

6.1.2. Pin

La clase Pin es el primer nivel de abstracción sobre la librería HAL y el código C con registros. Es una estructura de datos cuyo propósito es que los usuarios puedan identificar el Pin rápidamente en la ficha técnica y escribirlo directamente en el código sin tener que hacer traducciones ni convocar métodos. Esta pertenece a los Modelos de la ST-LIB Core.

Esta estructura se encarga de abstraer el concepto de los pines del microcontrolador en su totalidad. Lo primero que hace es mapear los valores de registro de los pines a unos valores más entendibles para el humano, usando los nombres que se le dan en la *nucleo* para ello. Por ejemplo, si a la configuración del pin A5 se accede con el registro **0x400000001802000000000020** permite al usuario obtener este registro dando los valores A y 5 a los **enum**⁶ **GPIOPin** y **GPIOPort**.

También mapea de la misma forma las posibles configuraciones que puede tener un Pin. Para eso utiliza hasta tres **enum** más; uno para saber si está ya configurado, otro para saber que configuración tiene, y uno tercero para indicar configuraciones especiales en caso de que los modos más comunes no sean lo que se busca.

Su última funcionalidad es la capacidad para registrar e iniciar cada uno de los pines a partir de los métodos **inscribe()** y **start()**. Estos métodos también añaden una capa de abstracción adicional ya que inscribe acepta directamente el nombre del pin y lo traduce a los valores necesarios para obtener su dirección de registro. Con este método se puede usar directamente **inscribe(A5,ANALOG)** en lugar de tener que obtener la dirección de memoria de la configuración usando A+5 y usar un writemem para introducir la configuración a mano. El método **start()** usa los valores introducidos en el inscribe para cada Pin para completar la configuración añadiendo todos los extras necesarios, como lo son DMA, relojes y la asignación de espacio de memoria para guardar las variables. Si **start()** se quedara sin relojes o canales DMA para asignar a los pines configurados usaría el ErrorHandler para avisar al programador, pero esta probabilidad es ínfima pues requeriría configuraciones muy específicas (*como ya explicaremos en la clase Time*).

⁶Un enum es una estructura de C que se dedica a enumerar distintos valores para distintos nombres, similar al concepto de una variable o un mapa pero se soluciona en tiempo de compilación, por lo que no afecta a la velocidad del programa

Como añadido adicional, existen contruidos ya todos los pines que hay disponibles en los micros de la familia H7 como estructuras de datos públicas, con los alias de `P+puerto+pin`. Por ejemplo, el pin A5 es accesible a través de un **extern**⁷ como **PA5**

6.1.3. DMA

El concepto de *Direct Memory Access*, acceso directo a memoria, o DMA viene de la necesidad en sistemas críticos de aliviar la carga del procesador. Como un porcentaje considerable del tiempo de ejecución es consumido gestionando los accesos a memoria y los datos recibidos a través de los periféricos, se decidió que sería más efectivo crear una unidad especial de asistencia al procesador en estas acciones antes que gastar recursos en aumentar más la potencia de este.

DMA es entonces la abstracción del uso de un hardware especializado que puede conectarse directamente a la memoria para transferir datos por un bus. En un principio solo hacía un acto muy primitivo, mover datos de un lugar a otro, en un rango de memoria limitado con la opción de ciclar dentro de este rango de memoria o terminar su ejecución y requerir reconfigurarse cuando llenase el buffer. Sin embargo, viendo lo efectivo y barato que era implementar los DMA, con los años se fue añadiendo más canales de acceso directo a memoria a los microcontroladores y se les dio mayor capacidad de cálculo, permitiendo incluso aplicar cálculos simples o relegar funciones a otras unidades de hardware y esperar su respuesta para guardarla dentro de lugares de memoria más específicos.

Mientras que el uso de DMA no es necesario para hacer nada, aprovechar esta tecnología puede mejorar mucho la eficiencia de los procesos del micro al poder ceder decenas de líneas de código que gestionan información de forma iterativa de la unidad de proceso a las unidades de DMA especializadas. Por ello, se necesitaba una abstracción de los canales DMA en la librería para poder aumentar su eficiencia en todos los periféricos que pudiesen hacer uso de esta.

El modelo de la DMA de la librería es bastante simplista, y deja mucha libertad y cosas por configurar a las clases de mayores niveles de abstracción, ya que esta utilidad es aplicable a una plétora de casos y no es estrictamente necesaria para ninguno de estos. Se dedica a definir con **enum** los canales de DMA, usar un mapa para guardarse los canales libres y los canales usados, y ofrecer un método de inscripción de DMA que asigna el primer canal libre o permite, en su lugar, elegir al programador uno específico. Luego tiene el método **start()** del que prácticamente todas las clases de la ST-LIB Core gozan y simplifica el inicio de estos servicios.

⁷extern es una palabra clave de C++ que permite indicarle al compilador que te refieres a una instancia de la variable que ya existe en otro documento, en lugar de querer crear una instancia nueva con el mismo nombre en el documento actual

6.1.4. Paquetes

La estructura de paquetes puede ser probablemente el módulo de la librería más difícil de comprender a primera vista. Debido a la complejidad de abstracción del concepto de paquete conservando su polimorfismo, el modelo packets usa templates⁸ infinitos recursivos para auto estructurarse en compilación dependiendo de las necesidades vistas en el código.

Como esta clase es tan compleja, está diseñada inicialmente como una “*caja negra*”, es decir, no hace falta entender como funciona para poder usarla. Simplemente, al declararse un paquete debe indicársele que tipos de variable guarda y cuantas, y luego usarse un método para o bien recibir un paquete desde un periférico y guardarlo dentro de la variable paquete creada; o bien meter valores dentro de la variable paquete (que también se puede hacer durante la construcción mediante su constructor), y enviar el paquete a través de uno de los periféricos con los servicios de comunicación que más adelante se verán.

Sin embargo, aunque no sea necesario comprender el módulo Packets para poder usarlo, en el resto de esta sección se trata de explicar como funciona. Si no se tiene comprender su funcionamiento o si se tiene una idea suficiente de su estructura con la explicación de arriba, puede pasar a la siguiente sección. En el caso de que quiera entender como funciona, la explicación comienza en el siguiente párrafo.

Como se ha mencionado arriba, el modelo de paquetes debe ser capaz de almacenar cualquier tipo o tipos de variable, en cualquier tipo de combinación, y guardarlos como un objeto paquete. A nivel de programación, esto quiere decir que debe ser capaz de recibir como parámetro cualquier combinación de tipos de variable, traducirlos a un valor binario para poder enviarlas a través de cualquier protocolo de información, y luego poder traducir de vuelta ese valor binario a los tipos de variable indicados para cerrar el concepto de comunicación por paquetes. Se requiere entonces que sea un modelo template para poder recibir cualquier tipo de variable que se pueda traducir a binario (*como a nivel de hardware todo está traducido a binario, se sabe que cualquier variable tiene al menos una forma de traducirla a binario*), y se requiere que pueda hacer *pattern matching* infinito, pues de antemano se desconoce cuantos parámetros van a darle al paquete.

Todos los métodos están sobrecargados para dos tipos de parámetros, o bien cualquier cantidad de parámetros mayor o igual que 1, o bien ningún parámetro. Este diseño está hecho a propósito para poder definir las funciones de los métodos como series geométricas, con su caso base (0 parámetros) y luego el caso de n parámetros. Al igual que las series geométricas, una función se define por una cantidad de operaciones más la misma función pero de n-1 valor (o parámetros, en este caso). Así, un paquete de 4 parámetros se define por la traducción del primer parámetro más la definición de un paquete que contenga los

⁸los templates son una palabra clave de C++ para funciones, variables y clases por igual que permite decirle al compilador que el objeto al que se le asigna funciona para más de un tipo de variable. Los templates pueden indicar que funciona para todas las variables numéricas, para todas las variables que guarden un valor, o para variables dentro de una clase definida por el usuario, permitiendo una gran expresividad y polimorfismo sin requerir centenas de líneas de código

otros tres parámetros. Se traduce el primer parámetro, y se convoca un paquete con tres parámetros que traduce el primero de sus parámetros y convoca a un paquete con dos parámetros. Así hasta llegar a 0 parámetros, donde se retorna el valor base (que es nada, pues el paquete está vacío) y cuando termina su convocación el paquete de un parámetro concatena su traducción a la del paquete de 0 parámetros, y termina también la convocación del paquete de un parámetro, llegando a hacer una cadena recursiva desde los parámetros requeridos hasta 0 y de vuelta a los parámetros requeridos. Básicamente, estos templates permiten hacer declaraciones recursivas en tiempo de compilación (luego en tiempo de ejecución también deberán hacerse convocatorias recursivas debido a que la palabra clave **inline** no funciona con los templates, pero solo será necesario en la construcción del paquete y en las traducciones)

Ahora pasaremos a explicar como maneja las transformaciones de las variables a binario. Aunque todas las variables tengan al menos una forma de transformarse a binario, esto no quiere decir que esta forma esté codificada para el uso del programa o que sea la forma de transformación más sencilla y eficiente. Por ello, la clase PacketValue sirve de soporte para la clase Packet y maneja los posibles valores que puede recibir como parámetros dependiendo de algunas propiedades que puedan requerir. Esta es una estructura de tipos que define grupos de tipos y dependiendo de a que tipo pertenezcan usa un método de traducción u otro. Los separadores de grupos definidos son **isContainer** y **isIntegral**, generando los grupos **Container**, **Integral** y **CustomButNotContainer**. Los que pertenecen al tipo **isIntegral** tienen una traducción directa a binario dada por C que está demostrada ser la más efectiva. Los que pertenecen a **isContainer** quiere decir que en verdad guarda más de una única variable, como una array, un mapa o un paquete (ofreciendo la opción de anidar paquetes dentro de paquetes).

6.1.5. Periféricos básicos: Analog y Digital

Analog y Digital en concepto son sencillos, y la mayoría del tiempo se pierde en configurarlos. Por ello, en la librería Core hemos creado las clases Analog, DigitalInput, DigitalOutput y EXTI principalmente para facilitar esto.

La estructura de los tres primeros es la misma: mapas que alocan los recursos de la placa y los metodos inscribe, start, y get value/set state que permiten configurar un pin (ya definido en la clase Pin) y recibir la id de la configuración, inicializar todos los pines, y hacer una orden de lectura/escritura en los periféricos. En este nivel aún se deben gestionar id's pero en la ST-LIB Low se pone una capa más de abstracción para tenerlo como un objeto de una clase.

Para utilizarlo es tan sencillo como parece, primero se convoca él **inscribe** dándole como parámetro el Pin que se desea configurar, y guardando su retorno en una variable uint8, luego se convoca el **start** y por último se toman las lecturas cada vez que se necesiten. Aquí un ejemplo del main:


```
uint8_t DigitalID = DigitalInputService::inscribe(PA5);
uint8_t AnalogID = ADC::inscribe(PA0).value(); #devuelve un optional
uint8_t OutputID = DigitalOutputService::inscribe(PC0);
HALAL::start(); #macro que activa todas las funcionalidades de la placa
DigitalInputService::start(DigitalID);
ADC::Start(AnalogID);
DigitalOutputService::Start(OutputID);
while(1){
    DigitalOutputService::set_pin_state(1);
    printf(ADC::get_value(AnalogID));
    printf(DigitalInputService::get_value(DigitalID));
}
```

Diagrama 11: Ejemplo de código con un lector análogo

Se recomienda utilizar el **HALAL::start()** en lugar de usar los starts de los módulos ya que algunos módulos dependen de que otros hayan iniciado y tienen que activarse en un orden en concreto. Por ejemplo, el módulo ADC usa las DMA para capturar los valores sin usar capacidad de procesador para mejorar la velocidad del código, pero tiene el pequeño defecto de no funcionar si las DMA no se iniciaron primero.

Aparte de estos tres sensores está el EXTI (External Interrupt) que permite activar interrupciones directamente a partir de cableado. El EXTI es más especial porque le puedes especificar que es lo que tiene que hacer cuando sea interrumpido a través de una *lambda expression*. Las lambda son básicamente funciones sin identificador, que se define por su cláusula de captura de parámetros, los parámetros que recibe, y el código a ejecutar.

```
uint8_t externalID = ExternalInterrupt::inscribe(PA5,() []{printf("Hola");}, FALLING);
HALAL::start();
```

Diagrama 12: Ejemplo de código con una interrupción externa GPIO

Este código define un external interrupt en el pin PA5 que imprimirá “Hola” cada vez que caiga la señal; es decir, que pase de 1 a 0. Puedes ponerle las opciones de **RISING** o **FALLING**, que quiere decir de 0 pasa a 1 y de 1 pasa a 0.

Hasta ahora nos hemos referido a la función **printf** como la función que imprime un texto, pero como el periférico por el que se imprime depende de la preferencia del programador, probablemente sea necesario sobrescribirla. Actualmente, en la ST-LIB lo imprime por UART, un protocolo de comunicación del cual hablaremos en las próximas secciones. Dependiendo del modo de programación (Nucleo o Board) estará en el uart1 o en el uart2 (definidos en el Runes)

La librería la sobrescribe en el módulo UART (sobrescribe el método **write** el cual es convocado por **printf**), así que para sobrescribirla debe removerse de UART primero.

6.1.6. PWM

El módulo PWM (*Power Width Modulated*) es un periférico especial que permite generar una señal periódica cuadrada con un periodo y un ciclo de trabajo ⁹ configurables. Estas señales son especialmente útiles para controlar motores, sincronizar comunicaciones y simular una señal analógica con cualquier voltaje deseado.

El módulo PWM tiene una clase con constructor, por lo que para usarla debe guardarse el objeto cuando haya sido declarado. Además del constructor, tiene los métodos **turn_on**, **turn_off**, **set_frequency**, **get_frequency** y **set_duty_cycle**. Estos métodos activan el PWM, lo desactivan, cambian la frecuencia (y con ello el periodo) con una uint dada, obtener la frecuencia que tiene guardada, y cambiar el duty cycle.

Los periféricos de esta clase no se inicializan en el **HALAL::start()** como protección, ya que entre otras cosas las PWM sirven como control de motor, e inicializarlo de forma discreta podría ser peligroso. En su lugar, el usuario debe activarlo y desactivarlo cuando vea necesario.

Además de la clase PWM, el módulo tiene tres clases que heredan de ella que añaden nuevas propiedades. Estas son PhasedPWM, DualPWM y DualPhasedPWM. PhasedPWM te permite añadir una fase angular a la onda cuadrada que genera el PWM, para sincronizar la onda con otras ondas a conveniencia. La DualPWM crea dos señales PWM negadas entre sí. Y la DualPhasedPWM es una DualPWM a la cual se le puede aplicar fase (ambas son afectadas por el cambio, ya que una es la negada de la otra). La fase es una float que puede ir desde -100 % hasta +100 %, que a nivel de onda representa un cambio de fase de -pi hasta +pi (en radianes). Un ejemplo rápido de uso:

```
DualPhasedPWM pwm(PA5,PA6);  
pwm.set_duty_cycle(40.0);  
pwm.set_frequency(1);  
pwm.set_phase(30.0);  
pwm.turn_on();
```

Diagrama 13: Ejemplo con dos PWM

Con este código hemos puesto dos PWM negadas entre sí en un duty cycle del 40 %, una frecuencia de 1 Hz, y una fase del 30 % ($+\pi \cdot 0.3$ de fase angular). Como se puede comprobar operar las PWM se ha vuelto mucho más sencillo, debido a que el módulo gestiona automáticamente toda la configuración de software.

⁹tiempo que se pasa activo dentro del periodo, en porcentaje. También se le conoce como *duty cycle*

6.1.7. Time

La clase time incluye la gestión de los relojes, las alarmas, y el rtc (*real time clock*). Como hay una cantidad limitada de relojes en el microcontrolador y algunos de estos son necesarios para hacer funcionar otros módulos como las PWM, en su lugar se utilizan tres relojes generales para que gestionen la gran mayoría de las alarmas y una interfaz con los relojes restantes en caso de querer una alarma crítica.

Los relojes generales low y mid están configurados para crear una interrupción cada 50 microsegundos y cada milisegundo, respectivamente. Luego de saltar la interrupción comprueba si las condiciones de las alarmas configuradas por el usuario se han cumplido, si es el caso ejecuta la lambda expression que le corresponde y en caso contrario la ignora. Un ejemplo de configuración:

```
uint64_t counter = 0;
Time::register_mid_precision_alarm(100, [&](counter){counter++;});
Time::register_low_precision_alarm(100, [&](counter){printf(counter);});
HALAL::start();
```

Diagrama 14: Ejemplo de código con dos alarmas

Este código sumará cada 100 microsegundos +1 al contador y lo imprimirá cada 100 milisegundos. Hay que indicar que el divisor del mid precision redondea hacia abajo y su paso es 50, así que poner menos de 50 microsegundos como intervalo de la alarma hará que no funcione correctamente. De la misma forma, si se pone 75 se redondeará a 50 para que encaje en el reloj.

Otro problema del mid_precision_timer es que requiere del reloj 23 de la placa; y dependiendo del código que se esté implementando se podría necesitar el reloj 23 para declarar PWMs en algunos pines. En el caso de la familia H7 hay 4 relojes de 32 bits (el 2, el 5, el 23 y el 24). El 24 se usa para el global timer, y el 2 y el 5 están asignados para high precision timers. Si se declaran los pines que utilizan el reloj 23 para el PWM y se usa el mid_precision_timer el iniciador del timer dará un error avisando de que no puede usarse el reloj para ambas cosas, pero se debe tener en cuenta a la hora de estructurar una placa. También se puede cambiar el Runes si la nucleo específica permite otra configuración, pero eso se tratará en su respectivo apartado. Como última medida se podría modificar el módulo time para usar uno de los relojes de los high precision timers y darselo al mid precision time, pero esto dejaría a la nucleo con solo un reloj de precisión crítica

Además de los timers de propósito general están los ya mencionados global timer y RTC. Estos sirven para mantener una cohesión temporal dentro del propio código; del orden de segundos y años respectivamente. Todos los relojes (y por tanto los timers) tienen un valor máximo antes de que hagan *overflow* en su contador y vuelvan a 0. Esto permite por un lado tener un activador por hardware para interrupciones sin necesidad de

implementar circuitería que aumente el precio (el bit de overflow). Pero por otro lado hace que se pierda toda la información sobre el tiempo que ha pasado cuando este reloj sufre overflow. Si han pasado 2 segundos o 1 mes es indistinguible para el reloj, ambos son un valor de 0 hasta 2^{32} , en periodos de reloj. Esto para cálculos temporales y comunicaciones es un gran problema, pues suelen requerir estos valores para poder funcionar. Por ello se usan estos dos relojes, cuyos plazos de *overflow* son mucho mayores para poder trabajar con ellos.

El global timer guarda en nanosegundos el tiempo que ha pasado desde que se inició, y tiene una capacidad de *overflow* de 16.84 segundos ($(2^{32})/(2,55 \cdot 10^8)$), que son el rango de su contador y su frecuencia). Lo bueno de este reloj es su gran precisión para cálculos críticos y su cercanía al hardware. Lo único malo es que los propios cálculos deben implementar su control de *overflow* (hay un ejemplo con el encoderSensor en la ST-LIB LOW, que lo usa) y que solo funciona para cálculos que trabajen en un rango de tiempo que no alcance las decenas de segundos. Si se quiere usar para estampas de tiempo más altas deberá hacerse un contador por encima que controle cuantas veces se ha hecho overflow, además de que su drift de reloj puede comenzar a acumularse.

El RTC (Reloj de tiempo real o Real Time Clock) es un reloj que guarda estampas de tiempo en el formato de fechas. Contiene contador (que se puede traducir a milisegundos), segundos, minutos, horas, día, mes y año; y su capacidad hasta el *overflow* es de 99 años, ya que tiene 8 bits alocados para guardar el año en BDC ¹⁰. El RTC se puede configurar para que use un generador de señales externo, para utilizar el cristal de cuarzo interno del micro, o para usar cualquiera de los 24 relojes como su generador de señales de reloj. De base está configurado en la librería para usar el cristal de cuarzo ya que tiene un desfase y un drift conocidos y documentados para todos los microcontroladores de la familia.

El RTC está pensado principalmente para hacer estampas de tiempo en comunicación entre placas o con clientes externos, y es especialmente útil ya que se puede sincronizar con otros agentes a través del protocolo ntp directamente. El rtc se inicia automáticamente con el st-lib start y está pensado para actualizarse utilizando ntp, pero tiene un método set_rtc para que se actualice desde código si la conexión por ethernet no es posible.

6.1.8. CORDIC

El módulo CORDIC es una librería que aprovecha la unidad de cálculo interna con el mismo nombre. Es una librería de aceleración de cálculo trigonométrico, la cual supera incluso a la librería arm math hecha para cálculo por estimación en arquitecturas de microcontroladores. La razón por la que esta pequeña unidad la supera es porque está diseñada específicamente para aplicar el algoritmo de Volder, especializado en funciones trigonométricas e hiperbólicas.

¹⁰BDC o Binary Coded Decimal es un formato alternativo a la int en el que se usan cuatro bits para expresar cada cifra de un número, haciendo que se escriba igual en decimal que en hexadecimal. Así por ejemplo, 13 se escribe 1101 0x0C en binario y 0001 (1) 0011 (3) o 0x13 en BDC.

La mayoría de las familias de microcontroladores STM incluyen la unidad de aceleración CORDIC, pero si el micro no lo incluye, esta librería no funcionará y se deberá desactivar el módulo dentro de la HAL (normalmente si tu microcontrolador no tiene CORDIC la flag de CORDIC MODULE está ya desactivada, apareciendo el código dentro de los `ifdef` en gris).

Una de las peculiaridades del CORDIC es que solo funciona con lógica binaria; es decir, no acepta coma flotante (ni floats ni doubles). El módulo CORDIC ya viene con un método que transforma las float (en radianes) a int32 en el formato que el CORDIC recibe. Sin embargo, no se recomienda abusar de este método ya que la transformación de int a float consume muchos recursos desde el punto de vista de la optimización (estamos hablando a nivel de ciclos de reloj), haciendo que vaya más lento incluso que la librería math arm con lógica de floats, lo cual le quita el sentido a usar el módulo CORDIC en primer lugar ¹¹.

Por ello CORDIC solo se puede usar de forma eficiente si tienes tres o más operaciones trigonométricas consecutivas que aplicar a un valor antes de tener que volverlo a transformar o si se usa lógica binaria en todo el código conectado a los cálculos trigonométricos. El primer caso se vería así:

```
double angle = pi * 0.5;
int32_t unary = RotationComputer::radian_f32_to_q31(angle);
int32_t *pointer1 = &unary;
int32_t *pointer2;
int32_t *pointer3;
RotationComputer::cos_and_sin(pointer1, pointer2, pointer3, 1);
RotationComputer::phase(pointer2, pointer3, pointer1, 1);
angle = RotationComputer::q31_to_radian_f32(unary);
```

Diagrama 15: Ejemplo de código usando CORDIC transformando las variables

El método **cos** and **sin** recibe uno o más ángulos a través de su primer parámetro y calcula el coseno y el seno, metiendo estos valores en los punteros out que recibe como parámetros. La razón por la que usa punteros es porque puede funcionar también con una matriz de valores. El último valor es, de hecho, el tamaño de la matriz que recibe. La otra forma de usarlo, y la más recomendable si tus periféricos te devuelven en int los valores a procesar (como suele ser el caso), es con la lógica de ints directamente:

Lo único tener en cuenta que hay que traducir el formato en el que se reciba los valores al formato unario para que el cálculo sea correcto. El formato de los ángulos recorre todos los valores posibles de las int, donde 180° es el valor máximo que puede tener la int y -180° el valor mínimo (máximo negativo).

¹¹hay que añadir que la familia H7 es la más potente de todas las familias, y la unidad CORDIC interna es la misma para todas las familias y su cálculo independiente del reloj del microcontrolador. Esto quiere decir CORDIC será más útil para otras familias y este módulo podría superar math arm incluso con recasting continuo en estas (porque un ciclo de reloj toma más tiempo y por ende CORDIC necesita menos ciclos de reloj para hacer la operación)

```
int32_t *values = values_array;  
int32_t *x;  
int32_t *y;  
RotationComputer::cos_and_sin(values,x,y,values_array_length);  
RotationComputer::phase(x,y,values,values_array_length);
```

Diagrama 16: Ejemplo de código CORDIC con lógica int

Esto tiene las ventajas de que aprovecha al máximo la precisión que le dan y que las sumas de ángulos no sufren *overflow*, ya que la representación de los ángulos negativos funciona igual que la de las ints. Por ejemplo, $\max \text{int} + 1$ es igual a $-\max \text{int}$, y $179,99995^\circ + 0,00001^\circ = 180,00005^\circ = -179,99995^\circ$.

El formato de las coordenadas simplemente representa un espacio acotado en -1 y 1 tanto en x como en y. Cuando superas el límite simplemente sufre *overflow*.

Los métodos **phase** y **modulus** operan en dicho espacio. **phase** mide los ángulos entre las rectas formadas por los puntos dados y el punto de origen con el vector (1,0). Esto es especialmente útil para comparar ángulos, ya que la fase entre ambos es “*la fase del primero con el eje x - la fase del segundo con el eje x*”. Oséa, se les hace fase a ambos vectores y se restan los resultados.

modulus sirve para obtener las distancias entre el origen y los puntos dados como valores escalares. Como esta distancia puede ser mayor que uno (el punto (1,1); por ejemplo), y como este número solo puede ser positivo, devuelve una uint32 en lugar de una int32 para poder llegar hasta el 2, cambiando su imagen de [-1,1] a [0,2]. Esto es importante porque **el modulus tiene sus propias unidades**, por lo cual no se puede utilizar sin más como unidad en el espacio. Si guardas el valor de retorno en una int en lugar de una uint y el resultado era mayor que 1 te devolverá un número negativo, lo cual es imposible pues las distancias (escalares) son positivas necesariamente.

6.1.9. Encoder

El encoder o codificador rotatorio es un periférico que traduce señales de pulsos a posición angular representada con un contador de giros, y es muy usado en el mundo de los motores y la electrónica para obtener posición y cambio de posición a partir del diámetro de la rueda y su posición angular. De este sale también el encoder lineal, que traduce las mismas señales a desplazamientos lineales directamente (los contadores representan una distancia igual a la distancia de las bandas del encoder lineal).

Los codificadores relativos son una versión más pequeña y económica que obtienen únicamente el cambio de posición relativo y no la posición angular absoluta. Simplemente, envían una señal “+1” o “-1” cada vez que una banda del encoder se cruza con el interruptor óptico, y suelen tener una precisión similar por una pequeña parte del precio.

El módulo encoder se encarga de transformar estos codificadores relativos en un codificador completo, guardando la posición inicial y sumando en un contador todos los cambios de posición para obtener el desplazamiento total. También se encarga de la configuración inicial del encoder como el resto de módulos. La declaración funciona así:

```
uint8_t encoder_id = Encoder::Encoder(PE0, PG1).get_value();
ST-LIB::start();
Encoder::turn_on(encoder_id);
while(1){
    uint32_t position = Encoder::get_counter(encoder_id);
    bool direction = Encoder::get_direction(encoder_id);
}
```

Diagrama 17: Ejemplo de código con el Encoder de la ST-LIB Core

Este código declara que el encoder está conectado a los pines PE0 y PG1, lo enciende y lee de forma continua tanto la posición como la dirección a la que fue la última rotación. Tener en cuenta que el encoder comienza su contador en 32768 para que comience lo más lejos posible del *overflow* y el *underflow*, debido a que dependiendo de la dirección puede sufrir uno u otro. En los cálculos que se hagan se ha de tener en cuenta que esta es la posición inicial del encoder. Además, si se tiene pensado usar en largas distancias se recomienda añadir lógica en caso de *overflow* (y *underflow*). En la ST-LIB Low ya se añaden todas estas abstracciones.

6.1.10. Flash

La mayoría de la memoria del microcontrolador es volátil, lo cual quiere decir que su información es destruida cuando la nucleo se reinicia o deja de alimentarse. Sin embargo, la mayoría de los microcontroladores, incluyendo todos los de la familia H7, tienen una pequeña memoria flash de 1 MB donde se puede guardar información no volátil como el *bootloader*, el código y variables o información generada durante el cómputo.

El módulo de la flash tiene como propósito manejar esta memoria de la forma más directa y simple posible. Tiene tres métodos: Read, Write y Erase. Los dos primeros reciben una posición de memoria inicial, un puntero con o bien los datos a escribir o el lugar donde se guardaran los datos leídos, y el tamaño de la lectura o escritura (cuantos bytes se va a leer). Erase por su lado simplemente recibe la posición inicial y la final y borra todo de por medio.

La flash se divide en páginas de 128 kB, y para escribir se debe acceder a una página de la flash en modo escritura, y cuando se accede de esta manera la memoria es borrada para poder escribir la nueva información. Para solventar esto simplemente se hace una lectura antes de la escritura, se anexiona la parte de la memoria que no se va a escribir al buffer de escritura, y se escribe toda la página.

6.2. ST-LIB Communication

La comunicación es en su mayoría parte de la ST-LIB Core, pero es tan extensa que por simplicidad es mejor darle su propio apartado. Al igual que la mayoría de la ST-LIB está apoyada en la librería HAL, y además parte de la ST-LIB Communication se apoya en la librería LWIP, un middleware que gestiona las comunicaciones por Ethernet TCP/IP muy eficiente y que trabaja principalmente con interrupciones.

ST-LIB Communication contiene los protocolos de comunicación más comunes en el mercado: *TCP (Ethernet)*, *UDP (Ethernet)*, *FDCAN*, *I2C*, *SPI*, y *UART*; además de introducir el protocolo de sincronización SNTP el cual está fuertemente vinculado con el reloj RTC.

Si algún protocolo de comunicación no es del interés del usuario, se recomienda desactivarlo eliminando del archivo main.h el define de ese módulo en la HAL o simplemente comentando del **HALAL::start()** y del **ST-LIB::update()** sus apariciones; pues estos protocolos tienen embebidas varias protecciones para asegurarse de establecer comunicación con largos márgenes de tiempo que pueden pausar al microcontrolador en ejecución.

Estas protecciones están activas para evitar falsas señales y asegurar de que no se pierde ningún mensaje, pero si no se están utilizando no son más que un estorbo para la ejecución del código. Especialmente los protocolos Ethernet pueden afectar al microprocesador, pues tienen una protección para no activar la placa hasta que un bus Ethernet viable sea conectado; por lo que el código no funcionará si no se le conecta el cable Ethernet o se desactiva el módulo Ethernet.

6.2.1. UART

UART es el protocolo de comunicación más sencillo que implementa la librería. Su funcionamiento es bastante básico, tiene un canal para el reloj que marca el paso para enviar y leer cada bit, y otro canal para enviar byte a byte la información.

El canal se encuentra en estado lógico 1 cuando está parado, y la comunicación se hace por paquetes de tamaño dependiente del estándar usado. En el caso del estándar 8N1 (*el más usado y la configuración inicial en la ST-LIB*) se inicia la comunicación con un bit en estado 0, se envía el byte de información (el tamaño del paquete en este estándar, 8 bits), y se cierra con un bit lógico 1.

En la librería UART tiene las opciones de comunicación de *polling* y por DMA con interrupciones, como la mayoría de módulos de la ST-LIB Core tienen. Las opciones son tan sencillas como leer y escribir, pues UART es un protocolo de comunicación punto a punto. El código por *polling*:

Este fragmento de código enviará continuamente los bytes 5, 5, 2, 2 y esperará a la respuesta desde el otro lado, la cual guardará en la matriz **data_receive** y volverá a


```
uint8_t uart = UART::inscribe(UART::uart2);
ST-LIB::start();
uint8_t data_send[] {5,5,2,2};
uint8_t data_receive[4];
while(1){
    if(UART::transmit_polling(uart, data_send)){
        UART::receive_polling(uart, data_receive);
    }
}
```

Diagrama 18: Ejemplo de código de UART usando polling

comenzar. Sobre el método **inscribe**, este recibe un periférico UART declarado en el módulo ¹² y definido en el archivo *runes* (en el ejemplo *uart2*). Como es común en la ST-LIB Core; él **inscribe** devuelve un número id que representa en que punto de su lista fue guardada la configuración del periférico.

Importante añadir que él **inscribe** solo funcionará si se hace antes del **ST-LIB::start**. Esto es debido a que el **start** activa todos los periféricos inicializados por el usuario; pero una vez el **ST-LIB::start** se ha ejecutado ya no activará más periféricos. Se puede circundar este problema con el método **start** de cada módulo, pero la mejor práctica es tener los periféricos fijos antes de activar el código de la placa.

La versión usando DMA tiene un par de añadidos nuevos, pero también añade complejidad al código. Hay dos métodos adicionales, *is_busy* y *has_next_packet*, que permiten saber si la comunicación por DMA ha terminado. El mismo código por DMA sería:

```
uint8_t uart = UART::inscribe(UART::uart2);
ST-LIB::start();
uint8_t data_send[] {5,5,2,2};
uint8_t data_receive[4];
while(1){
    if(!UART::is_busy(uart)){
        UART::transmit(uart, data_send);
    }
    if(UART::has_next_packet(uart)){
        UART::receive(uart, data_receive);
    }
}
```

Diagrama 19: Ejemplo de código UART usando DMA

En este caso el código hará una petición a la DMA para que trasmita los datos del **data_send** siempre que está este libre, mientras que al mismo tiempo revisará si ha

¹²El módulo UART solo tiene declarados diez periféricos uart (de *uart1* a *uart10*). Si se quieren añadir más, se deberá modificar el header del módulo para declararlos

recibido un paquete del otro lado y de ser el caso lo guardará en **data_receive**.

En este caso no tiene mucho sentido usar la DMA porque no hay más código que ejecutar que comunicarse por UART, pero la ventaja viene en que mientras está gestionando el canal de comunicaciones por DMA el microcontrolador puede seguir corriendo código.

Un ejemplo muy básico de cuando podría servir la DMA con UART es si se tiene más de un canal UART. El código sería así:

```
uint8_t first_uart = UART::inscribe(UART::uart1);
uint8_t second_uart = UART::inscribe(UART::uart2);
ST-LIB::start();
uint8_t data_send[] {5,5,2,2};
uint8_t data_receive[4];
uint8_t data_receive2[4];
while(1){
    if(!UART::is_busy(first_uart)){
        UART::transmit(first_uart, data_send);
    }
    if(!UART::is_busy(second_uart)){
        UART::transmit(second_uart, data_send);
    }
    if(UART::has_next_packet(first_uart)){
        UART::receive(first_uart, data_receive);
    }
    if(UART::has_next_packet(second_uart)){
        UART::receive(second_uart, data_receive);
    }
}
```

Diagrama 20: Ejemplo de código con dos UARTs

En el caso de *polling* el programa tendría que esperar a que terminara la comunicación por el primer bus antes de comenzar a comunicarse por el segundo; y aún más grave, tendría que esperar a la respuesta de ambos buses antes de comenzar la siguiente comunicación o ejecutar cualquier otro código. Con DMA, sin embargo, no hay ninguna espera y se pueden mantener comunicaciones paralelas por varios buses.

6.2.2. I2C y SPI

Tanto I2C como SPI son dos protocolos de comunicación basados en el paradigma de controlador - trabajador. En estos protocolos el controlador marca la velocidad de reloj (que en ambos casos es un canal físico que marca cada cuanto se envía un bit de información), y actúa de forma proactiva haciendo peticiones que esperan una respuesta.

Por contraparte, los trabajadores se adaptan a la velocidad de reloj indicada (mientras no supere su capacidad máxima) y son unos agentes reactivos; esperan una petición para comenzar el proceso o devolver información del resultado

En ambos protocolos es posible tener más de dos dispositivos conectados en una única

red de comunicación; y en I2C además es posible tener más de un controlador que arbitre el reloj y haga peticiones. Por ello, para identificarse entre los distintos dispositivos estos protocolos añaden utilidades adicionales que UART no tiene.

En el caso de SPI, usa un cable específico para cada trabajador que lo activa o desactiva. El I2C por su parte utiliza un sistema de ip de 7 bits básico, y cada vez que se transmite una orden, esta incluye la ip emisora y la receptora.

Quitando estas diferencias a nivel de la ST-LIB SPI y I2C funcionan igual que UART, y de hecho en el momento de la creación de estos módulos se usó UART como base para hacerlas.

```
uint8_t first_spi_worker = SPI::inscribe(SPI::spi1);
uint8_t second_spi_worker = SPI::inscribe(SPI::spi2);
ST-LIB::start();
uint8_t data_send[] {5,5,2,2};
uint8_t data_receive[4];
uint8_t data_receive2[4];
while(1){
    SPI::chip_select_on(first_spi_worker);
    SPI::chip_select_off(second_spi_worker);
    SPI::transmit(first_spi_worker, data_send);
    SPI::receive(first_spi_worker, data_receive);
    SPI::chip_select_on(second_spi_worker);
    SPI::chip_select_off(first_spi_worker);
    SPI::transmit_and_receive(second_spi_worker, data_send, data_receive2);
}
```

Diagrama 21: Ejemplo de código SPI con dos trabajadores

Como se puede en el código de la figura 21 funciona por *polling*. La razón de ello es que los trabajadores que se comunican por SPI (o I2C) no avisan de cuando han terminado (pues podrían interrumpir otra comunicación). En su lugar, su controlador debe preguntarles y estos le devolverán un valor u otro dependiendo de si acabaron su proceso o aún están ocupados.

Como las DMA no tienen la suficiente capacidad lógica para resolver si la información que han recibido implica que el periférico ha terminado o aún está en proceso, no tiene mucho sentido utilizar las DMA en estos protocolos.

Adicionalmente, su comportamiento ya ofrece en cierta forma paralelización, pues el controlador sabe cuando va a devolverle la información que desea (sea esta información la respuesta o que aún está procesando); ya que los trabajadores solo responden cuando se les pide, y conservan la respuesta hasta entonces. Debido a esto, el controlador puede seguir ejecutando código sin temer a perder la respuesta de su petición y recibirla cuando las tareas críticas hayan sido atendidas.

Como I2C puede usar los mismos pines para comunicarse con toda la red de trabajadores, no hay necesidad de declarar un I2C por cada uno de ellos. Además, al requerir una

comunicación por paquetes algo más compleja se ha hecho un módulo de apoyo. Quitando estos dos matices, funciona de la misma forma que el protocolo SPI. Aquí un pequeño fragmento de código del I2C: ¹³

```
uint8_t i2c_workers = I2C::inscribe(I2C::i2c2, 0x10);
uint8_t first_worker_id = 0x40;
uint8_t second_worker_id = 0x50;
ST-LIB::start();
uint8_t data_send[] = {0xAA, 0x00, 0x00};
uint8_t data_receive[3];
I2CPacket first_worker_order(first_worker_id, data_send);
I2CPacket second_worker_order(second_worker_id, data_send);
I2CPacket first_worker_read(first_worker_id, data_receive);
I2CPacket second_worker_read(second_worker_id, data_receive);
while(1){
    I2C::transmit_next_packet(i2c_handler_id, &first_worker_order);
    I2C::transmit_next_packet(i2c_handler_id, &second_worker_order);
    I2C::receive_next_packet(i2c_handler_id, &first_worker_read);
    I2C::receive_next_packet(i2c_handler_id, &second_worker_read);
}
```

Diagrama 22: Ejemplo de código I2C con dos trabajadores

Las id tanto del controlador como de los trabajadores pueden ponerse tanto en formato hexadecimal como en formato decimal, pero lo más común en el mercado es lo primero, por ello está puesto de esta forma en el ejemplo.

6.2.3. CAN-FD

CAN-FD^[7] es un protocolo algo más avanzado que goza de múltiples sistemas de recuperación de información y usa dos canales negados para detectar interferencias. Esto lo hace más robusto y le permite alcanzar mayores velocidades; pero también incrementa sus requisitos mínimos para poder funcionar, en el sentido del hardware.

Además, CAN-FD; a diferencia de los protocolos previamente mencionados, sí está protegido por derechos de autor y es propiedad de la empresa privada Bosch. Esto tiene sus ventajas y desventajas, como un mejor soporte pero una mayor dependencia.

Si se desea comunicar CAN-FD a un sistema que esté usando un RTOS (sea sé un ordenador o un microprocesador), lo más probable es que se requiera de una interfaz PCAN-USB.

En cuanto a la interfaz de software del módulo CAN-FD, está estructurada para funcionar de la misma manera que los otros protocolos de comunicación. Como siempre, en la figura (23) se mostrará un ejemplo de código.

¹³En algunas versiones de la librería se requiere añadir el tamaño del paquete pues aún no se aplicaron los cambios para usar span

```
uint8_t fdcan = FDCAN::inscribe(FDCAN::fdcan1);
ST-LIB::start();
uint8_t send_message_id = 10;
uint8_t receive_message_id = 12;
uint8_t data_send[] = {0xAA, 0x00, 0x00};
uint8_t data_receive[3];
FDCAN::packet packet_read(data_receive, 12, FDCAN::DEFAULT);
while(1){
    FDCAN::transmit(fdcan, message_id, data_send, FDCAN::DEFAULT);
    FDCAN::read(id, packet_read);
}
```

Diagrama 23: Ejemplo de código FDCAN

Este código se ha desenrollado un poco por claridad, pero la idea es que se pueda hacer *hard-code* con la mayoría de valores dentro de los paquetes y que estos sean la referencia a la que acudir. Como se ve, en el envío se usa un *array* y la recepción un paquete. Lo cierto es que en este módulo son intercambiables, y además se puede utilizar cualquier otra estructura de información que se agrupe dentro del módulo *span* de C++. El último valor del **transmit** y del **packet** es el tamaño de paquete.

6.2.4. UDP

UDP es el protocolo más sencillo de Ethernet. Se basa en mandar paquetes de información sin ningún tipo de comprobación o protección a una o más ip. No hay garantía de que el paquete llegue, ni es ese el propósito de este protocolo.

La idea de este protocolo es poder enviar información recurrente o un gran flujo de datos continuo, consumiendo la menor cantidad de proceso posible. Si el bus Ethernet está montado correctamente y no se dan condiciones extremas, la probabilidad de pérdida de paquete es ínfima; así que para procesos que requieran una actualización continua con un margen de error aceptable este protocolo es ideal.

Por otra parte, para sistemas críticos que requieren enviar una señal específica con la máxima prioridad, probablemente se desee establecer una conexión TCP en su lugar.

Algunos ejemplos de uso para UDP serían:

- Actualización de datos en tiempo real
- Grabación u obtención de imágenes de cámara en directo
- Comunicación por voz a través de la red
- Sincronización de relojes

Mientras que UDP en sí no tiene ninguna protección, el módulo Ethernet de la librería sí tiene unas mínimas protecciones en marcha; pensadas para que no reduzcan la efectividad

de UDP.

La librería comprueba primero; antes de activar los sensores, periféricos y actuadores de la placa; si el bus de Ethernet es viable. Esta comprobación se hace a través del protocolo ARP, que permite resolver usando las direcciones MAC la ip de cada dispositivo conectado a la red.

El protocolo ARP solo puede fallar si hay más de un dispositivo con la misma MAC en la red o si hay solo 1 dispositivo conectado a la red, o lo que es lo mismo, solo un lado del cable Ethernet está conectado a un dispositivo activo. El módulo Ethernet de la librería solo falla al activar la placa (se queda esperando infinitamente) si el protocolo ARP no se resuelve o si la ip escrita en el módulo ya está ocupada.

Debido a que lo más probable es que el usuario desee conectar más de una placa a la misma red, es de interés cambiar tanto la MAC como la IP de la placa antes de conectarla la comunicación Ethernet fallara.

Esto es tan sencillo como ir dentro del proyecto a LWIP/app/lwip.c y cambiar dentro del método **MX_LWIP_Init** la matriz **IP_ADDRESS** para la ip; y LWIP/Target/ethernetif.c dentro del método **low_level_init** la matriz **MACAddr** a la dirección MAC deseada.

```
IPV4 ip_placa("192.168.0.4");
IPV4 ip_objetivo("192.168.0.5");
uint16_t puerto = 50000;
ST-LIB::start();
DatagramSocket connection(ip_placa, puerto, ip_objetivo, puerto);
uint8_t send_message_id = 10;
uint8_t receive_message_id = 12;
uint8_t data_send[] = {0xAA, 0x00, 0x00};
uint8_t data_receive[3];
StackPacket packet_send(send_message_id, data_send);
StackPacket packet_read(receive_message_id, data_receive, [&]() { data_send[2]++; });
while(1){
    connection.send(packet_send);
    ST-LIB::update();
}
```

Diagrama 24: Ejemplo de código UDP

Este código envía un paquete que contiene **data_send** a la ip objetivo, y cuando la ip le responda activará una interrupción que ejecutara la lambda dentro del **packet_read**.

Primero de todo, se puede ver que ahora se ha comenzado a usar el **ST-LIB::update()**. Este método se encarga de actualizar en tiempo real las acciones automáticas de la librería; que incluyen protecciones, detección de errores y obtención de paquetes. En protocolos más sencillos capturar los paquetes era suficientemente sencillo como para que el usuario pudiese gestionarlo sin resultar un problema, pero debido a la velocidad y exigencias del protocolo Ethernet, además de su gran variedad de uso; se ha decidido que la librería lo

gestionase por el usuario.

Si no se ejecuta a suficiente velocidad el **ST-LIB::update()**, el buffer de Ethernet puede llenarse y perderse información (dependiendo del tráfico de la red puede rondar en el orden de horas o de segundos), por lo que se recomienda ejecutar el **ST-LIB::update()** lo más rápido posible (estando solo en el bucle **while** es una ejecución en el orden de microsegundos).

Si se siguen buenas prácticas de programación en microprocesadores (como tener interrupciones de muy poco código que activen marcadores para ejecutar otras cosas en futuro, ejecutar métodos con límites de tiempo y haciendo múltiples pasos en distintas llamadas), no debería haber problemas ni en los entornos con más carga de comunicación.

Además, se puede ver que el paquete de envío **packet_send** no tiene ninguna función lambda. Esto es meramente para que no provoque interrupción cuando reciba un paquete con esa id, e ignore los mensajes con esta id. Pero perfectamente se podría tener un paquete que tanto se enviara como se pudiese recibir desde otros dispositivos.

Por último, se le puede dar cualquier función de tipo *void*, no es necesario que sea una lambda. De hecho, aunque las funciones lambda sean más cómodas pueden provocar una reducción de velocidad si se usan continuamente, y puede ser de interés evitarla si se encuentran problemas de velocidad de ejecución.

6.2.5. TCP

El protocolo TCP es el otro módulo mayor que surge de Ethernet. A diferencia de UDP, TCP tiene decenas de protecciones y siempre que se tenga un bus viable en Ethernet y una conexión establecida entre los dos puntos de comunicación, siempre llegará el mensaje al punto deseado.

El único problema con TCP es que es más lento que el protocolo UDP, especialmente en el caso de que se desee enviar miles o millones de paquetes por segundo, pues TCP requiere de una confirmación (conocida como *ACK*) para cada uno de los paquetes, y además todos ellos deben tener cuarenta bits de cabecera obligatoria para que la comunicación funcione, aunque este contenga solo dos bits.

Por ello, TCP se recomienda para paquetes críticos para el funcionamiento del sistema, de un solo envío, o de información que se debe recopilar con la máxima precisión.

Quitando la necesidad de establecer una conexión, el módulo TCP de la ST-LIB funciona exactamente igual que el módulo UDP, así que se recomienda leerse también el apartado de UDP para comprender correctamente el funcionamiento de TCP, ya que hay cosas que no se explicarán en este apartado para evitar repetición.

El primer ejemplo de código, mostrado en la figura 25, muestra que la conexión como usuario funciona prácticamente igual que UDP. La única diferencia es que la construcción del *socket* puede fallar si no se consigue establecer conexión con el servidor en el tiempo

```
IPV4 ip_placa("192.168.0.4");
IPV4 ip_objetivo("192.168.0.5");
uint16_t puerto_cliente = 50500;
uint16_t puerto = 50000;
ST-LIB::start();
Socket connection(ip_placa, puerto_cliente, ip_objetivo, puerto);
uint8_t send_message_id = 10;
uint8_t receive_message_id = 12;
uint8_t data_send[] = {0xAA, 0x00, 0x00};
uint8_t data_receive[3];
StackPacket packet_send(send_message_id, data_send);
StackPacket packet_read(receive_message_id, data_receive, [&]() { data_send[2]++; });
while(1){
    connection.send_order(packet_send);
    ST-LIB::update();
}
```

Diagrama 25: Ejemplo de código TCP como usuario

límite, normalmente debido a que el servidor no esté activado aún.

```
IPV4 ip_placa("192.168.0.4");
IPV4 ip_objetivo("192.168.0.5");
uint16_t puerto = 50000;
ST-LIB::start();
ServerSocket connection(ip_placa, puerto);
uint8_t send_message_id = 10;
uint8_t receive_message_id = 12;
uint8_t data_send[] = {0xAA, 0x00, 0x00};
uint8_t data_receive[3];
StackPacket packet_send(send_message_id, data_send);
StackPacket packet_read(receive_message_id, data_receive, [&]() { data_send[2]++; });
while(1){
    if (connection.is_connected()){
        connection.send_order(packet_send);
    }
    ST-LIB::update();
}
```

Diagrama 26: Ejemplo de código TCP como servidor

En este segundo ejemplo de código, mostrado en la figura 26, la placa hace de servidor. Por como está estructurado, por ahora cada ServerSocket solo puede aceptar una conexión. Si se requieren de múltiples conexiones, la única opción actualmente es usar múltiples ServerSocket. En un futuro se tiene pensado hacer una abstracción en la ST-LIB Low que maneje estos problemas, pero para la mayoría de necesidades en el mundo de los microprocesadores esto suele ser más que suficiente.

6.2.6. SNTP

El protocolo SNTP es el protocolo de sincronización de relojes usado en la ST-LIB. Se basa en comunicación UDP y funciona de forma completamente automática en la ST-LIB. Funciona a través de conectarse a un servidor indicado en SNTP.cpp (que se puede cambiar por cualquier otro servidor accesible desde la red cambiando el valor de la variable **TARGET_IP**).

Si no se desea cambiar la ST-LIB, también se puede activar de forma manual con otra ip usando el método **sntp_update** justo después del **ST-LIB::start()**, que actualizará la lista de servidores de las que puede reclamar la hora añadiendo la ip introducida y removiendo la de la ST-LIB.

Siempre que el servidor NTP esté funcionando, actualizará la hora una vez al activarse la placa y una vez cada hora después de que la placa se active. El valor de la hora obtenido se guardará dentro del RTC, un submódulo dentro de time hecho especialmente para el funcionamiento del módulo SNTP, que puede guardar no solo un valor de reloj, si no además una hora y fecha.

Para obtener todos los valores del RTC, es tan fácil como usar el método adquirente del RTC, como se muestra en la figura 27

```
ST-LIB::start();  
while(1){  
    RTCData time_and_date = Time::get_rtc_data();  
    print(RTCData.year);  
}
```

Diagrama 27: Ejemplo de código SNTP imprimiendo el año actual

Al usarse este código se podrá comprobar que al principio imprimirá fechas extrañas. Esto es debido a que hasta que no resuelva por primera vez la comunicación SNTP la fecha no va a estar actualizada, cosa que se debe tener en cuenta al programar el código.

Como advertencia adicional, SNTP no funcionará si la distancia temporal de los relojes del servidor y el cliente (el microcontrolador) están a una distancia mayor de 36 años. Actualmente el reloj RTC está configurado para activarse en el 1-1-2023, pero si no se activa adecuadamente o la distancia temporal a 2023 es mayor de 36 años (sea por la razón que sea), el protocolo no funcionará.

6.3. ST-LIB Low

La ST-LIB Low es la capa intermedia entre la ST-LIB Core y la ST-LIB High; y es la capa que contiene la mayor cantidad de abstracciones, cambiando la forma de programar los microcontroladores de lenguaje C con manejo de registros e id de periféricos a una estructura más parecida a C++, usando POO y múltiples macros declaradas en **ST-LIB::start** y **update**.

Por lo general, la mayoría de las cosas que un usuario utilice estarán en esta capa, pues la ST-LIB Core sufre de ser de muy bajo nivel; mientras que la ST-LIB High puede llegar a ser demasiado específica.

Todo lo que se encuentra en la ST-LIB LOW no es más que una abstracción de la ST-LIB Core para facilitar su uso, así que no es estrictamente necesaria. No obstante, está diseñada para ser más cómoda y rápida de usar añadiendo el mínimo coste de proceso posible, por lo que se recomienda usarla por encima de la ST-LIB Core siempre que sea posible.

6.3.1. Counter

Counter es un pequeño módulo de utilidad que sirve para contar las ocurrencias de un evento en una estampa de tiempo dada y obtiene su frecuencia, en Hz.

Para cualquier cálculo que requiera del uso de frecuencia (o de periodo, que es la inversa de este), Counter puede ser útil. En la figura 28, un pequeño ejemplo:

```
Counter contador(100);  
ST-LIB::start();  
while(1){  
    contador.count();  
}
```

Diagrama 28: Ejemplo de código Counter que cuenta el número de veces que se pasa por el bucle While, por segundo

Este código ejemplo obtiene la frecuencia con la que se pasa por el bucle **while**. Esto puede ser extremadamente útil para poder analizar el código, pues muchas veces los requisitos funcionales de un código de microcontrolador dado se definen en mínimo de veces por segundo.

Una vez se tenga el bucle de código que se quiere comprobar en marcha (sea un bucle o una alarma que se repite continuamente), una forma muy sencilla de comprobar que se cumplen los requisitos es poner dentro del método en cuestión la función count y obtener la frecuencia con la que se ejecuta. Esto permitirá ver si realmente se cumple el concepto del código o hay algo afectando a la ejecución.

Muchas veces puede haber una colisión o una condición de carrera entre dos métodos que se deban ejecutar a una frecuencia dada, sea porque tardan más de lo debido o porque suceden las interrupciones que lo activan al mismo tiempo. Esta herramienta está para identificar esos casos y separar la teoría de la práctica.

Además de sus utilidades de depurado puede servir para hacer cálculos basados en fórmulas físicas para hacer un predictor.

6.3.2. Stopwatch

StopWatch es la contraparte de Counter, y hace la función de cronómetro; pero en lugar de ofrecer su información en unidades métricas la ofrece en ciclos de reloj. Para usarlo, simplemente se inicia una instancia de StopWatch donde se quiera comenzar a contar y se usa el método Stop para terminar de contar y recibir el resultado de la diferencia.

Al igual que Counter, StopWatch sirve como herramienta de depurado de código. StopWatch ofrece en nanosegundos la diferencia de tiempo desde que se inició hasta que se para, permitiendo analizar cuanto tiempo le cuesta al microprocesador ejecutar uno o más métodos.

```
Counter contador(100);
ST-LIB::start();
while(1){
    Stopwatch::start("mysw");
    contador.count();
    Stopwatch::stop("mysw");
}
```

Diagrama 29: Ejemplo de código de Stopwatch que cuenta cuanto tiempo ocupa el contador

En el código de la figura 29 se muestra como funciona su uso. A diferencia de la mayoría de la ST-LIB LOW no usa clases, y esto es debido a que una vez se termina el uso de StopWatch debe eliminarse de memoria.

Para evitar que el usuario tenga que gestionar la memoria y pueda sufrir fugas¹⁴, se ha decidido que funcionara como una librería estática, pero en lugar de usar id numéricas recibe una string que hace de pseudónimo para el reloj. El valor “mysw” es meramente el nombre que se le ha designado a ese StopWatch, y cualquier string es un valor válido.

StopWatch también sirve para hacer cálculos físicos o predictores que requieran diferencial de tiempo en sus fórmulas. El error en la diferencia de tiempo que devuelve StopWatch es menor que $\pm 2 / (\text{velocidad de reloj del microprocesador})$ segundos si no se tiene en cuenta la deriva de susodicho reloj.

¹⁴la fuga de memoria es cuando el código compilado pide continuamente espacios de memoria sin liberarlos hasta que toda la memoria se llena, el puntero apunta a un valor inexistente, y la ejecución termina con un error.

6.3.3. DigitalOutput

DigitalOutput es el módulo de la ST-LIB LOW que abstrae el módulo Core con el mismo nombre. Su objetivo es hacer más fácil el uso de los GPIO para los usuarios, encargándose de activar los periféricos cuando se active la placa, gestionar las id de forma interna, y de añadir nuevas funcionalidades.

Los métodos de DigitalOutput son el constructor, **turn_on**, **turn_off**, **set_pin_state**, y **toggle**. Como sus nombres indican, permiten crear el objeto, encender, apagar, cambiar el estado, e invertir el estado actual; respectivamente. Aquí un ejemplo de código:

```
DigitalOutput out(PA5);
ST-LIB::start();
while(1){
    out.toggle();
}
```

Diagrama 30: Ejemplo de DigitalOutput que hará cambiar de estado lo más rápido posible al pin PA5

Este código cambiará el estado del pin out lo más rápido que pueda. No se recomienda usar este código con ningún led, o en general con ningún componente eléctrico sensible, pues cambiar tan rápido de estado podría desgastarlo rápidamente. Un código más razonable sería que hubiese una espera al final del bucle; aunque podría provocar problemas al estar perdiendo valioso tiempo de proceso en una espera vacía.

El mejor código para un caso así sería introducir el **toggle** dentro de una interrupción que ocurriese periódicamente, usando; por ejemplo, el módulo time:

```
DigitalOutput out(PA5);
ST-LIB::start();
Time::register_low_precision_alarm(1000, out.toggle);
while(1){
    /*dejamos el while para que el programa no termine*/
}
```

Diagrama 31: Ejemplo de DigitalOutput que hará cambiar de estado al Pin PA5 cada 100 ms

6.3.4. ErrorHandler

El ErrorHandler es el módulo de retroalimentación de usuario por excelencia de la librería. Funciona usando la función printf (que la librería por defecto sobrescribe para que use el módulo UART) para enviar no solo el error que se le introduzca, sino además información adicional de depurado como el lugar en el que sucedió dentro del código. En el código ejemplo de la figura 32 primero se dará un error en la línea debajo del **start**; y

```
ST-LIB::start();  
ErrorHandler("Ha habido un error");  
while(1){  
    ErrorHandler("Ha habido un error");  
}
```

Diagrama 32: Ejemplo de ErrorHandler

luego se darán continuamente errores en la línea debajo del bucle. Importante notar, que aunque el usuario haya introducido la misma cadena de caracteres en ambos errores, serán distinguibles gracias a la información adicional que provee el ErrorHandler.

Una cosa importante del ErrorHandler es que está vinculado a muchos módulos superiores de la librería. Uno de ellos es el ProtectionManager, un módulo especial que se encargará de cambiar el estado de la placa y su funcionamiento cuando se dé una situación en la que el código previsto ya no pueda recuperarse a un estado funcional.

El ErrorHandler está vinculado de base al ProtectionManager, y cuando salte activará todas las protecciones de este. Así que se recomienda usar el ErrorHandler únicamente para errores.

6.3.5. Math

El módulo Math trata de abstraer el módulo CORDIC de la ST-LIB Core, gestionando los casos marginales, evitando divisiones por cero, y añadiendo nuevas funcionalidades que se puedan calcular con suficiente velocidad para que sea razonable usarla en lugar de arm_math.

Math posee las mismas funciones que CORDIC más la tangente y arco tangente, en los métodos tg y atg. Hay algunas funcionalidades de CORDIC que no se pueden acceder desde Math directamente por simplicidad; como poner múltiples peticiones de cálculos en una sola invocación del método. Este es un compromiso que toma el módulo Math para mantener su uso simple a cambio de perder algo de versatilidad y eficiencia en situaciones específicas.

Esta pérdida de velocidad aumenta la duración de un ciclo del bucle que haga múltiples cálculos iguales con distintos valores en alrededor de un 13% (depende de la operación específica), pero quitando el caso concreto en el que se desee hacer centenas de veces el mismo cálculo sin operaciones intermedias, la velocidad de Math es muy similar a la de CORDIC; e incluso en este caso supera math_arm con lógica de int en un factor medio de 2.5 (dependiente de la operación específica).

En cuanto al uso de Math, es tan sencillo como convocar la función y recibir el valor que retorna:

```
int32_t angle = pi/3*MAX_INT_VALUE; /* pi/3
int32_t tan = 0;
radians*/
ST-LIB::start();
while(1){
    tan = Math::tg(angle);
}
```

Diagrama 33: Ejemplo de uso de tangente con Math

Como se puede ver ya no hace falta reservar punteros a las variables necesariamente, lo cual es una de las ventajas de no poder recibir más de un valor por llamada. Como estos punteros no son necesarios y en lugar se recibe el valor en el retorno de la función, se pueden anidar llamadas.

```
int32_t angle = pi/3*MAX_INT_VALUE; /* pi/3
int32_t tan = 0;
radians*/
ST-LIB::start();
while(1){
    tan = Math::tg(Math::atg(Math::tg(angle)));
}
```

Diagrama 34: Ejemplo de uso de múltiples llamadas con Math

6.3.6. Sensors

El grupo de módulos Sensors cumplen todos la misma función: abstraer las lecturas de periféricos con el uso de clases. Hay un caso especial, el del encoder, que trataremos en su propia sección.

DigitalSensor es el primero de estos módulos y, como su nombre indica; codifica un sensor digital. Su funcionamiento es sencillo: Creas el objeto antes del **STLIB::start** con una referencia a la variable que se desee modificar, y después cuando se desee se convoca la función **read** que actualiza el valor de susodicha variable.

Como normalmente estos valores son accedidos por muchas funciones al mismo tiempo se ha evitado que devolviese en el **read** el valor mencionado para que el usuario no convocara múltiples veces la función de lectura **read** para obtener el mismo valor; y redujera accidentalmente la velocidad de su código. En su lugar, se debe decidir a que velocidad se actualiza el valor del puntero y acceder a este, que contiene siempre el valor más actualizado, siempre que se desee; para guiar al programador a estructurar de forma correcta su código, pues aunque este sea más abstracto sigue siendo código de microcontrolador.

El siguiente módulo del grupo es LinearSensor, que trata de codificar un sensor análogo

pero con la abstracción añadida de escalar y desplazar el valor para ponerlo en las unidades adecuadas dentro del puntero. Sí se desea obtener el voltaje, meramente con poner el escalado **slope** a 1 y el desplazamiento **offset** a 0 se conseguirá en voltios la señal recibida.

LinearSensor sigue las mismas guías de estructura que DigitalSensor, y fuerza al usuario a acceder al valor de puntero directamente en lugar de obtener el valor del **read** para evitar que abrume con demasiadas peticiones de lectura al controlador. Añadir que LinearSensor es una clase que hace uso de templates para funcionar con el tipo de variable preferida por el usuario (normalmente una variante de la coma flotante)

Otra abstracción de lectura de periféricos básica es el LookupSensor, que tiene como objetivo abstraer las *Lookup Table*, tablas de valores preprocesados con el objetivo de transformar una fórmula matemática de alto coste de procesado a un único acceso a memoria a cambio de tener que ocupar dicha memoria.

Su funcionalidad es sencilla: recibe una pequeña matriz que representa la tabla de valores de un tamaño deseado por el usuario; y separa el rango de valores posibles que puede recibir en análogo (normalmente de 0 a 3.3V, se puede cambiar con una variable denominada REFERENCE_VOLTAGE) entre todas las entradas de la matriz. En la figura 35 un ejemplo de estos tres tipos de Sensor, funcionando todos al mismo tiempo

```
PinState sensor_state = OFF;
float sensor_voltage = 0;
float table[] = {1,10,100,1000}
float table_lecture = 0;
DigitalSensor dsensor(PA5, &sensor_state);
LinearSensor lsensor(PA6, 1, 0, &sensor_voltage);
LookupSensor tsensor(PA0, table, &table_lecture);
ST-LIB::start();
while(1){
    dsensor.read();
    lsensor.read();
    tsensor.read();
    printf(table_lecture);
}
```

Diagrama 35: Ejemplo de uso de DigitalSensor, LinearSensor y LookupSensor

Este ejemplo guarda el estado de PA5 en **sensor_state**; el voltaje de PA6 en **sensor_voltage**, y el valor de la *look up table* en **table_lecture**. Importante pasar los valores por referencia (&) o no los actualizará. Además, imprime el valor obtenido por la LookupTable accediendo directamente a la variable, como se ha mencionado previamente.

En el ejemplo de la figura 35 **Table_lecture** contendrá el valor 1 si el voltaje se encuentra entre [0,0,825), el valor 10 si se encuentra entre [0,825,1,65), y así hasta llegar a 3.3 v; que es el previamente mencionado valor de referencia.

SensorInterrupt es una versión especializada de DigitalSensor que además abstrae Ex-

ternaInterrupt, activando una interrupción siempre que el estado del sensor cambie de la forma indicada en su constructor.

De base está configurado para que la interrupción se active siempre que cambie de baja a alta (*RISING*), pero las opciones de alta a baja o ambas también están disponibles. Cuando esta interrupción se active, la función que se pasó en su constructor se ejecutará (a menos que otra interrupción de mayor importancia este en marcha, en cuyo caso se encolará). Un ejemplo de código del SensorInterrupt:

```
PinState sensor_state = OFF;
counter = 0;
SensorInterrupt isensor(PA5, [&]() { counter++; },
&sensor_state, ExternalInterrupt::FALLING);
ST-LIB::start();
while(1){
}
```

Diagrama 36: Ejemplo de uso de SensorInterrupt

Como se puede comprobar en este fragmento de código, en ningún momento se hace una lectura. Sin embargo, la interrupción sucede aunque no se esté actualizando el valor; pues esa es la idea de la interrupción. Hay que tener en cuenta que no necesariamente todos los pines tienen la capacidad de usar EXTI por lo que puede fallar si se usa un pin sin esta habilidad. Ante la duda, se recomienda revisar la ficha técnica del microcontrolador o usar el .ioc para revisar si está entre las opciones.

6.3.7. EncoderSensor

El EncoderSensor es el módulo que abstrae el Encoder, transformando el contador en un medidor de posición, velocidad y aceleración. Este módulo se estructura de la misma forma que el resto de módulos Sensor, pero debido a su complejidad y sus requisitos funcionales se ha decidido dar un apartado entero para este.

Para calcular la derivada de la posición en el tiempo se ha usado una aproximación del cálculo conocida por su eficiencia, la aproximación por diferencia finita regresiva [8]. Dentro del código, esto significa guardar en una matriz múltiples puntos separados por una distancia en el tiempo; a los que dentro del código se les referirá como *frames*, y obtener la diferencia entre los dos, dividirla entre la distancia en el tiempo; y el resultado es la derivada aproximada.

Esta aproximación es finita porque produce el resultado de un cálculo hecho dentro de un marco finito, es diferencial porque produce sus valores a partir de la diferencia entre dos puntos respecto a una variable (en este caso el tiempo); y es regresiva porque solo usa el punto que se quiere medir y los anteriores, nunca los posteriores.

No se usan los valores posteriores debido a que esto requeriría un predictor y se ha comprobado en pruebas que el intercambio de ganancia de precisión por pérdida de velocidad de código no es razonable; y de ser necesario hay otras medidas más efectivas que tomar antes del uso de un predictor.

Esta fórmula es extremadamente rápida y tiene el potencial de ofrecer una precisión cercana $\pm \text{paso}/2$ donde el paso es el desplazamiento necesario para que el encoder complete una rotación de 360° en su señal PWM. Sin embargo, requiere de calibrado de las variables para conseguir este nivel de cálculo.

Las variables a modificar para el calibrado son: **COUNTER_DISTANCE_IN_METERS**, **N_FRAMES**, y **FRAME_SIZE_IN_SECONDS**. La primera de las variables, **COUNTER_DISTANCE_IN_METERS**; define cuanta distancia real representa cada paso del encoder, lo cual dependerá del ángulo que este represente y del perímetro de giro (en caso de no ser lineal). Para ello, se deberá consultar la ficha técnica del encoder específico que se esté usando y del dispositivo que se trate de medir.

N_FRAMES y **FRAME_SIZE_IN_SECONDS** son, por otra parte, variables que afecta a como se captura el valor de cálculo. **N_FRAMES** representa cuantos *frames* guarda el EncoderSensor antes de sobrescribirlos, y **FRAME_SIZE_IN_SECONDS** representa la distancia mínima entre dos *frames* para que sean guardados. Aumentar cualquiera de las dos reduce el efecto de las perturbaciones y aumenta la precisión; pero también aumenta el tiempo de medida absoluta. Esto es, el tiempo que tarda desde que el objeto real a cambiado de una velocidad A a una velocidad B en actualizar la medida dada por el programa completamente de A a B. Este tiempo es, exactamente, $(\text{N_FRAMES} - 1) * \text{FRAME_SIZE_IN_SECONDS}$.



Diagrama 37: Grafica de ejemplo de una curva de posicion tiempo

En el diagrama 37 se puede observar un ejemplo de una gráfica generada a partir de la posición del objeto de medida en el tiempo, estando la posición en el eje y (metros) y el tiempo en el eje x (segundos). Asumiendo que su trayecto termina en el punto 0.1 m, y por ende en el segundo diez está parado; el cálculo de velocidad del EncoderSensor tardará el tiempo de medida absoluta en indicar, desde el segundo diez, que nuestro objeto se ha parado completamente.

Asumiendo un **N_FRAMES** igual a 3 y un **FRAME_SIZE_IN_SECONDS** de 1.0 s, en el segundo 10 indicará 0.125 m/s, en el 11 0.085 m/s, y en el 12 0.0 m/s. Como se puede observar, no alcanza la velocidad real hasta que han pasado dos segundos. Este efecto se puede reducir decrementando el valor de cualquiera de las dos variables, pero esto también aumentará el efecto de perturbaciones y reducirá la precisión de medida.

Adicionalmente, aumentar **N_FRAMES** ocupa más espacio en memoria (pues aumenta el tamaño de la matriz de guardado), y aumentar **FRAME_SIZE_IN_SECONDS** provoca mayor retardo en la actualización de los valores, pues estos solo se actualizan cada nuevo *frame*. Unos valores razonables rondan **N_FRAMES** 500 y **FRAME_SIZE_IN_SECONDS** 0.0007, pero esto realmente depende de los requisitos del usuario.

Una vez calibrado, a la hora de programar el código es igual de simple que el resto de la familia de módulos Sensor. Se declara antes del **STLIB::start()**, se activa ¹⁵ y se toman medidas de forma cíclica.

```
float position = 0;
float speed = 0;
float acceleration = 0;
float direction_vector = 0;
EncoderSensor encode(PC6,PC7,&position,&direction,&speed,&acceleration);
ST-LIB::start();
encode.start();
while(1){
    encode.read();
}
```

Diagrama 38: Ejemplo de uso de EncoderSensor

Lo único de relevancia que queda indicar en este código es que la variable **FRAME_SIZE_IN_SECONDS** solo funciona mientras se ejecute el método **read()** del EncoderSensor con un periodo de tiempo entre ejecuciones estrictamente menor que **FRAME_SIZE_IN_SECONDS**. En el caso contrario, el código seguirá funcionando pero en lugar de usar **FRAME_SIZE_IN_SECONDS** para la distancia entre *frames*, usará el tiempo entre ejecuciones pues es en estas donde se obtienen los *frames*.

Cada *frame* guarda el punto en el tiempo en el que fue calculado, así que el cálculo

¹⁵el EncoderSensor no se activa solo en el start porque comienza a tomar medidas en ese punto, que considera el segundo 0. Como cuando se desee que comience a medir y tome de marco de referencia, el segundo 0 depende del usuario, se ha optado por no activarlo con el **STLIB::start()**

no se verá afectado directamente si hay algún retraso, pero si de forma consistente se ejecuta con menor velocidad de la requerida por su **FRAME_SIZE_IN_SECONDS**, se puede comprometer el tiempo de medida absoluta y la velocidad de actualización de los valores.

Se recomienda poner el **read()** o bien en el bucle permanente del código (como se ve en el diagrama 38) o bien en una interrupción periódica para asegurar su correcto funcionamiento.

6.3.8. StateMachine

StateMachine es un módulo de abstracción del concepto de una máquina de estados, un concepto muy usado en el mundo de la programación de microcontroladores pues permite separar los requisitos funcionales entre varios estados; y añadir un funcionamiento especial para cuando el producto se encuentre en un estado irrecuperable por código, un estado de emergencia conocido generalmente como *FAULT*.

Para poder usarlo, primero se deben de crear los posibles estados en los que el código se puede encontrar. Estos se pueden definir con números, con variables *uint8*, o bien con enumeraciones de C++ (este último es el más recomendado meramente por orden de código). Una vez se tienen los estados de la máquina, se debe crear la propia máquina de estados e insertar estos estados.

```
enum states{
    INITIAL,
    OPERATIONAL,
    FAULT
}
int main(){
    ST-LIB::start();
    StateMachine principalStateMachine();
    add_state(INITIAL);
    add_state(OPERATIONAL);
    add_state(FAULT);
    while(1){
        ST-LIB::update();
    }
}
```

Diagrama 39: Ejemplo de la declaración de una máquina de estados

En el código del diagrama 39 se puede ver como se vería un *main* que incluyese una máquina de estados con tres estados: **INITIAL**, **OPERATIONAL** y **FAULT**. Una vez se tienen los estados, se deben introducir las transiciones.

Para introducir las transiciones hay dos posibles opciones. La primera es crear un

método que compruebe una condición y en caso de cumplirse fuerce la transición con el método **force_change_state()** de la clase `StateMachine` y convocarlo cada cierto tiempo con una interrupción, dentro de otro método, o en el bucle. La segunda opción es crear una función que devuelva verdadero cuando se cumpla una condición, y usar el método **add_transitions()** para añadir una transición de un estado A a otro B cuando se cumpla la condición. Esta comprobación se hace dentro del método **check_transitions()**; que también pertenece a la clase y se deberá convocar periódicamente.

Lo más común es usar la segunda opción, pues permite controlar en que estados se comprueba esta condición, a que estados llevará cuando se active, y permite centralizar en una única llamada el control de la máquina de estados, ayudando a organizar el código.

Sin embargo, la primera opción tiene también sus casos de uso, pues permite crear cambios de estado más críticos, que se comprueben aparte del resto de transiciones y fuercen inmediatamente el cambio de estado nada más se cumplan.

```
enum states{
    INITIAL,
    OPERATIONAL,
    FAULT
}

int main(){
    ST-LIB::start();
    StateMachine principalStateMachine();
    add_state(INITIAL);
    add_state(OPERATIONAL);
    add_state(FAULT);

    add_transition(INITIAL, OPERATIONAL, start_success);
    add_transition(INITIAL, FAULT, start_failure);
    add_transition(OPERATIONAL, FAULT, fault_values);

    Time::register_high_precision_alarm(100,[&](){
        if(more_important_fault_values()){
            principalStateMachine.force_change_state(FAULT);
        }
    });

    while(1){
        ST-LIB::update();
        principalStateMachine.check_transitions();
    }
}
```

Diagrama 40: Ejemplo de como añadir transiciones a una máquina de estados

Para evitar abarrotar el código con demasiadas líneas en el diagrama 40, no se ha escrito la definición de las funciones **start_success**, **start_failure**, **fault_values**, o **more_important_fault_values**; pero todas estas són funciones que retornan un valor

booleano, que deberá definir el propio usuario dependiendo de sus requisitos funcionales.

Dentro del mencionado diagrama, se puede observar tanto la primera forma de definir transiciones, las cuales han sido insertadas dentro de una interrupción que se cumplirá cada 100 microsegundos; como la segunda forma, que se comprobará cada vez que el bucle permanente se ejecute, y se define a través de los **add_transitions**, los estados que deben transicionar, y la función que retorna verdadero si se cumplen las condiciones.

En este punto ya se tiene una máquina de estados funcional, de la cual se puede substraer el estado en cualquier instante a través de la variable **current_state** dentro del objeto, y siempre que las comprobaciones se hagan correctamente se obtendrá el estado deseado. Sin embargo, el módulo no termina aquí, pues tiene funcionalidades añadidas; y está diseñado para que la mayoría del código se ejecute dentro del objeto StateMachine.

Se puede añadir código que se ejecute inmediatamente tras una transición, código que se ejecute periódicamente a través de una alarma únicamente en ciertos estados, y anidar máquinas de estados a estados de otra máquina de estados que solo se actualicen y ejecuten sus funciones mientras su máquina padre esté en el estado al que fueron anidadas; sin límite de niveles de anidamiento.

```
enum states{
    INITIAL,
    OPERATIONAL,
    FAULT
}

int main(){
    ST-LIB::start();
    StateMachine principalStateMachine();
    add_state(INITIAL);
    add_state(OPERATIONAL);
    add_state(FAULT);

    principalStateMachine.add_enter_action(entry_fault, FAULT);
    principalStateMachine.add_enter_action(entry_operational, OPERATIONAL);
    principalStateMachine.add_exit_action(exit_initial, INITIAL);
    principalStateMachine.add_low_precision_cyclic_action(
        cyclic_operational,
        std::chrono::milliseconds(10),
        OPERATIONAL
    );

    while(1){
        ST-LIB::update();
        principalStateMachine.check_transitions();
    }
}
```

Diagrama 41: Ejemplo de como usar las herramientas de StateMachine

En el código del diagrama 41 se ha removido las transiciones y no se han definido los métodos `entry_fault`, `entry_operational`, `exit_initial`, y `cyclic_operational` para evitar, de nuevo, abarrotar el código con líneas. Las transiciones se pueden observar en el diagrama 40 y los cuatro métodos tienen un código arbitrario, definido por el usuario, que han de ejecutar cuando se cumpla la condición a la que se vincularon y no retornan nada (*return type void*) obligatoriamente.

Además, se puede ver que ahora se usa la librería de C++ `chrono` para introducir la frecuencia con la que se deben ejecutar las acciones cíclicas. Lo único que hay que apuntar sobre esto es que las acciones cíclicas siguen teniendo las mismas limitaciones que las alarmas del módulo `Time`, por lo que hacer que una acción cíclica de baja precisión se ejecute cada menos de un milisegundo, una media precisión cada menos de 50 microsegundos, o una alta precisión cada menos de 1 microsegundo; no cumplirá con las estampas de tiempo indicadas pues está fuera de su rango de control.

6.4. ST-LIB High

ST-LIB High es la capa de mayor abstracción de toda la librería, y usarla puede traer grandes beneficios de seguridad, diseño y velocidad de desarrollo; pero también puede añadir un coste de ejecución alto.

En este apartado se mostrarán los módulos de uso general dentro de la ST-LIB High, pues los módulos más específicos son un proyecto a futuro y están preparadas para cubrir a un público más concreto.

6.4.1. Notification

Notification es el módulo de abstracción de los paquetes TCP de la librería. Esta abstracción está diseñada para simular la difusión masiva a partir de comunicar a todas sus conexiones TCP las notificaciones recibidas.

Además, las Notification están diseñadas para comunicar cadenas de caracteres y su objetivo es hacer llegar al servidor software que se comunique con el proyecto información en un formato más amigable para las comunicaciones escalables; siendo un ejemplo JSON.

Su construcción es igual que la de un paquete TCP, con la ligera variación de que el *callback* es obligatorio y los resultados los aloja dentro del propio objeto en lugar de en un puntero externo que se le deba pasar en el constructor.

En el ejemplo solo hay una conexión disponible, pero funcionará con tantas conexiones como se declare, y tanto con conexiones servidor como con conexiones cliente. Es importante apuntar, que al igual que con los paquetes, solo se puede recibir una notificación que tenga una id de paquete declarada; lo cual está hecho para evitar comunicaciones y comportamientos indeseados.

Un paquete con id 13 no sería recibido por este código; mientras que uno con id 10 sería recibido correctamente, aplicaría el efecto de una puerta negada a la variable booleana

```
ST-LIB::start();  
bool value = false;  
Notification not_and_spread(10,[&]() {value = !value;});  
Notification spread(20,nullptr,"spread me");  
ServerSocket incoming_communication(IPV4("192.168.1.4"),50500);  
while(1){  
    ST-LIB::update();  
    if(value){  
        spread.notify();  
    }  
}
```

Diagrama 42: Ejemplo de como usar Notify

value, y esparciría usando difusión amplia a todas sus conexiones el valor. El paquete con id 20 simplemente esparciría la cadena de caracteres recibida sin hacer nada más. **Notify()** es el método utilizado para comunicar la notificación a todas las conexiones TCP abiertas, y sirve para comenzar la difusión del mensaje.

Se recomienda no hacer directamente un **Notify()** dentro del método *callback* de una notificación, pues puede provocar un efecto cascada que abrume el bus Ethernet con infinidad de paquetes si no se gestiona correctamente.

6.4.2. ProtectionManager

El ProtectionManager es el módulo supervisor de la librería, que mantiene el control de las máquinas de estados y permite controlarlas con mayor facilidad.

Como se mencionó en el ErrorHandler, ProtectionManager tiene algunas protecciones internas ya programadas para casos que siempre las requieren, siendo el caso más interesante la activación del ErrorHandler. Siempre que se haga una llamada al ErrorHandler y el ProtectionManager esté activo, el ProtectionManager mandará a *FAULT* (o el estado de emergencia que el usuario le haya dado) inmediatamente a la máquina de estados.

Además, cuando la máquina de estados llega al estado de emergencia, envía una notificación que todas las placas que usen el módulo ProtectionManager poseen, y manda su máquina de estados también al estado de emergencia; provocando que todas las placas de la red vayan al estado de emergencia para poner en modo seguro a todo el código del producto en caso de una emergencia.

En el ProtectionManager hay dos tipos de protecciones, las normales y las de alta frecuencia. El propósito de esto es facilitar la organización de las dos formas de crear transiciones en las máquinas de estados para el caso del ProtectionManager. La única diferencia entre ambas es que una se convoca en el método **check_protections**, y la otra en **check_high_frequency_protections**. La frecuencia a la que estos métodos se

convoquen depende del propio usuario.

Las protecciones en sí funcionan de forma distinta que las transiciones de estado, y no planean necesariamente sustituirlas. En su lugar, son una forma de poner límites al estado del código, como un máximo tiempo, un rango de aceptabilidad, o que una variable sea igual que un estado. Esto las hace mucho más rápidas de ejecutar y permite estructurarse de forma limpia los estados en los que las variables se pueden encontrar.

Adicionalmente, cuando se activa el ProtectionManager la notificación que envía explica brevemente en la cadena de caracteres que guarda que variable alcanzo el límite y cuando la alcanzo¹⁶. El errorHandler tiene un mensaje específico que da toda la información que este ofrece normalmente por el **printf**.

```
enum states{
    INITIAL,
    OPERATIONAL,
    FAULT
}

int max_is_ten = 0;

int main(){
    ST-LIB::start();
    StateMachine principalStateMachine();
    add_state(INITIAL);
    add_state(OPERATIONAL);
    add_state(FAULT);
    ProtectionManager::link_state_machine(principalStateMachine, FAULT);
    ProtectionManager::set_id(15);
    add_protection(&max_is_ten, Boundary<int, ABOVE>(10));
    Time::register_low_precision_alarm(1, ProtectionManager::check_protections());
    while(1){
        ST-LIB::update();
        principalStateMachine.check_transitions();
    }
}
```

Diagrama 43: Ejemplo de como usar el ProtectionManager

En el diagrama 43 se puede ver un ejemplo de como se usaría el ProtectionManager para proteger una variable y controlar cuando esta alcanza su máximo valor. Las opciones son **BELOW**, **ABOVE**, **OUT_OF_RANGE** (recibe tanto límite inferior como superior), **EQUALS**, **NOT_EQUALS**, y **TIME_ACCUMULATION**. El caso especial **ERROR_HANDLER** no está pensado para que sea usado para otra cosa.

¹⁶usando RTC, por lo que para esto requiere un servidor NTP o tener el reloj correctamente sincronizado

7. Testeo y perfilado de la librería

7.1. Perfilado

El perfilado es el arte de analizar el rendimiento de un fragmento de código a partir del uso de múltiples herramientas. Para hacer estas pruebas se usó el modo depurado de la placa, los módulos de la librería especializados para obtener tiempos y ocupación de espacio, e incluso bucles infinitos con contadores para ver la frecuencia de ejecución con la mayor fidelidad posible.

El mayor problema de este arte es que al tratar de obtener información sobre el código, de una forma u otra se debe adulterar su ejecución, sea añadiendo contadores, poniendo modos especiales que tratan de controlar los relojes y la pila de ejecución, o añadiendo métodos que ocupan espacio.

Aunque se haya hecho perfilado en todos los niveles de la librería, se ha tenido especial precaución con la ST-LIB Core y las herramientas de mayor uso. Para el *benchmarking*, se ha comparado nuestros métodos con los métodos que poseen las mismas capacidades más famosos en el mercado de los microcontroladores stm32.

Estos son, *HAL*, *LL*, *lwip*, y *arm-math* principalmente. Como la *HAL* posee muchos ejemplos de código para cada uno de sus módulos, se han comparado estos directamente contra un código con las mismas capacidades con nuestra librería. Para los otros dos simplemente se cambiaron líneas de nuestro código de perfilado por las líneas equivalentes de estas librerías, pues carecen de ejemplos tan extensos o completamente funcionales.

Lo primero de todo es el tiempo de inicio. Este tiempo es el que más preocupaba en cuanto a la ST-LIB, pues las abstracciones usadas, para evitar que en ejecución tuviesen un coste de procesamiento adicional se añadieron estructuras que se definían al comienzo de la librería, en la activación de la ST-LIB.

Esta preocupación no es en vano, pues en el mejor de los casos, usando todos los módulos; la ST-LIB puede tardar casi 100 milisegundos en iniciarse, obligando a tener un estado adicional de inicialización para asegurar consistencia en la ejecución del sistema completo (asumiendo que hayan múltiples placas conectadas en un solo sistema o red).

En el peor de los casos puede tardar más de cuatro segundos en lanzar un error de conexión faltante, aunque en este caso sigue siendo mejor que la *HAL*, que en caso de error simplemente se queda atascada en un bucle infinito al lanzar un *HARD_FAULT*. El caso que preocupa es en el de funcionamiento correcto, pues el coste de inicio de la *HAL* es varias veces menor.

Esta diferencia era esperable, pues toda abstracción tiene un coste, y la mayoría de los costes de la ST-LIB se han desplazado a tiempo de iniciación o tiempo de compilado. Como nota, el tiempo de compilado es de media seis veces más con la librería, debido a que compila dos veces; en C y C++, además de compilar dos proyectos, la librería y el código ejecutable. Este factor se reduce más cuanto más código se compile.

El tiempo de compilado no es algo de excesiva importancia en el campo de los micro-controladores, pero la librería puede alcanzar los cincuenta segundos compilando el código de una placa de control[9] en un portátil de gamma media alta¹⁷ desde cero¹⁸, y en una OrangePi 5 con ubuntu 22.04 y las herramientas de compilado arm puede llegar a tardar diez minutos.

En cuanto a los tiempos de ejecución, cada módulo tiene sus propias peculiaridades. Los Pines, DMA, PWM, Encoder y los periféricos básicos no tienen ninguna diferencia de coste notable (<5 %) respecto de la HAL, pues realmente hacen lo mismo que sus homólogos, pero con más configuración inicial para dejar las herramientas a mayor disposición del usuario.

Time es el peor de los casos al ser usado con una única alarma que cambia el estado de una variable puede ocupar un 70 % más dentro de la interrupción. Aunque este número parezca excesivamente alto, hay que tener en cuenta que esta diferencia se mide en ciclos de reloj; debido a que el código de la HAL para interrupciones provocadas por una alarma se puede compilar en unas veinte líneas de código ensamblador, incluyendo las líneas de cambio de contexto.

Al comparar CORDIC con su versión de la HAL eran equivalentes, pues se copió y pego el código para ahorrar un cambio de contexto. A pesar de que el documento de uso de CORDIC de stm32[10] afirma que la diferencia de velocidad de CORDIC y arm-math con lógica de int alcanza un factor de 10 de diferencia, en la práctica solo se ha conseguido un factor de 2.5 sin transformación a coma flotante.

Esto es debido a varias razones. La primera es que su código no usa funciones que abstraigan el uso de CORDIC. Están insertados completamente dentro del bucle en el que se ejecutan. Luego el test que hicieron fue con la familia M4 y no con H7, y por último en su prueba compilan con otra versión del compilador de arm y en un modo distinto (aunque ambas pruebas utilizan el modo de máxima eficiencia).

Sin embargo, incluso al imitar sus pruebas lo más posible (usando la familia H7 a 255MHz), y usando el perfilador de su IDE, no hemos alcanzado más que un factor de tres en casos específicos en la diferencia entre arm-math con lógica de enteros y CORDIC con lógica de enteros. Sí que superaba por un factor de diez a la lógica de coma flotante, aunque la transformación los iguala si se quiere tener en cuenta los casos límites. Si se hace asumiendo que no se alcanzaran los límites, le supera por un factor de 2.

Sin embargo, comparar arm-math y CORDIC no es completamente justo, pues la primera es más consistente y no tiene casos límites mientras que la gestión de los casos límites de la segunda podría, si no se estructura correctamente, ocupar decenas de veces más tiempo de proceso. Al fin y al cabo, estas librerías tratan sus operaciones en el orden de ciclos de reloj, rondando los doscientos por operación incluyendo cambios de contexto

¹⁷procesador intel i7 de 9ª generación, 16GB RAM, gráfica 1050Ti para portátiles, ejecutando en disco SSD Kingston de 500GB con 50 GB de memoria restantes

¹⁸El compilado en C++ puede usar antiguas compilaciones para reducir el tiempo de compilaciones futuras hasta en un 90 %

para el primero, y los sesenta para CORDIC.

Por ello, se ha decidido comparar `arm-math` con el módulo `Math` de la `ST-LIB`, que usa el módulo `CORDIC`, pero se encarga de gestionar todos los casos límite, como que `CORDIC` no devuelva valores correctos cerca de los 180° o pierda precisión cerca del cero.

Para seno, coseno, y fase es 10 % más lento que `CORDIC` (2.3 veces más rápido que `arm-math`), la tangente y la arcotangente es un 50 % más rápida que `CORDIC` (las cálcula usando seno y coseno en lugar de pedirlos directamente), y el módulo es un 30 % más lento que `CORDIC`, pues es el que más casos límites sufre. Se ha probado con todos los valores posibles dentro de un entero en C++ comparándolo con `arm-math` y la diferencia es menor del 0.5 % en todos los casos¹⁹

Las comunicaciones apenas se han probado en el perfilado, pues el cuello de botella es siempre el bus, y muchas veces la `DMA` gestiona la mayoría de la carga de proceso; haciendo que no ocupen prácticamente tiempo. Para las que no tienen `DMA` en la `ST-LIB` (`I2C` y `SPI`), sin protecciones y contra polling tienen el mismo coste. Con protecciones, la `ST-LIB` ocupa el doble de tiempo cuando estas no se activan nunca y hasta siete veces más tiempo cuando se activan una vez por transacción (hay un fallo por cada mensaje).

Sin estas protecciones, cuando hay un fallo y se intenta comunicar una vez más toda la placa se va a `HARD_FAULT` así que trabajar sin las protecciones es inviable. La `HAL` requiere que estructures tus propias protecciones, y se dedica simplemente a avisarte de si ha habido un error, no a gestionarlo.

Por lo general, si se usa la `ST-LIB Core` la diferencia de procesado ronda del 5 % al 15 %²⁰ más de tiempo para esta respecto a la `HAL`; pero este es el caso solo si no se aplica ninguna protección a la ejecución de la `HAL`, para lo cual no ha sido diseñada. La idea de la `HAL` es dejar al usuario estructurar sus propias protecciones y formas de recuperarse de los errores, por lo que usarla así, aunque posible, es inviable para un proyecto que tenga la consistencia o seguridad como requisitos.

Con las protecciones mínimas para un funcionamiento robusto, la diferencia funcional en la práctica y para la familia `H7` entre la `ST-LIB Core` y la `HAL` es nimia, en todos los ámbitos menos tiempo de compilado y de inicialización. A menos que esos dos parámetros sean decisivos para el diseño del proyecto, no hay razón para usar la `HAL` en lugar de la `ST-LIB Core` directamente.

Contra `LL`, la librería que hace competencia a la `HAL` en el ámbito de los microcontroladores `stm32`, sí que hay una diferencia más notable. En algunas funciones específicas se puede optimizar hasta reducir un 25 % el tiempo de ejecución. Y de hecho, en algunos casos la `ST-LIB` esconde funciones que hacen uso de registros al estilo `LL`, atravesando la `HAL` completamente.

Sin embargo, observando la diferencia de legibilidad, reusabilidad, y versatilidad de ambos códigos (44, 45), se puede comprobar que para la mayoría de casos de uso usar la

¹⁹menos en los límites de la tangente ya que `Math` de la `ST-LIB` devuelve un valor especial que usa como infinito

²⁰depende de que recursos de la librería use el código

LL directamente es inviable, y se suele terminar por crear una capa de abstracción por encima, que termina por hacer lo mismo que la ST-LIB; aunque más personalizado para el problema en concreto, lo cual puede ser interesante para las empresas que tengan los recursos para gastar en proyectos como este.

```
while ((SPI1->SR & SPI_SR_TXE) == 0){}
SPI1->DR = (uint8_t)n;
while ((SPI1->SR & SPI_SR_RXNE) == 0){}
while ((SPI1->SR & SPI_SR_BSY) != 0){}
return SPI1->DR;
```

Diagrama 44: Ejemplo de código que envía un valor por un periférico SPI usando LL directamente, sin protecciones ni control de bus

```
I2C::transmit_next_packet(i2c_handler_id, &byte_order);
```

Diagrama 45: Código del diagrama 44 usando ST-LIB, con las protecciones añadidas

Por supuesto, la creación de una librería de abstracción especializada para las necesidades del proyecto específico siempre será una opción, pero aunque a nivel de utilidad y funcionamiento sea superior a la ST-LIB (siempre que se personalice correctamente); muchas veces el coste de tiempo, recursos e inversión monetaria necesarios para la producción de tal librería no merece la ganancia o incluso puede ser prohibitivo para muchos proyectos, y no sirve como una solución a corto o medio plazo.

7.2. En la práctica

Una de las mejores formas de probar un producto es en un caso de uso real, y cuanto más general y completo sea este caso, mejor será la información obtenida en esta prueba. Como el objetivo principal de la ST-LIB es apoyar la producción de proyectos como el vehículo Hyperloop, Kenos es la mejor plataforma de pruebas posible para la librería.

Kenos dispone de ocho microcontroladores distintos con una plétora de funcionalidades. Desde uso de predictores para controlar levitación electromagnética a través de un motor LIM, hasta comunicaciones con un servidor externo para obtener información sobre el funcionamiento de todo el sistema en vivo. El código de todas las placas se puede encontrar públicamente en los repositorios de la organización HyperloopUPV dentro de Github. [11]

Dentro del vehículo van montadas las placas de control de baterías (BMSL, BMSH y OBCCU), bootloader (BLCU), control del vehículo (VCU), control de propulsión (PCU) y control de levitación (LCU). Fuera, en la infraestructura que soporta los raíles se encuentra la TCU, encargada de gestionarlo.

En los códigos de estas placas se puede comprobar, primero de todo, la versatilidad y facilidad de uso de la librería. Partiendo desde un template de trabajo, como puede serlo el project-template; Con alrededor de 1000 líneas se puede implementar un sistema seguro de control de trabajadores que hace uso de varios *PID* para gestionar todos los sistemas a los que está conectado, a la vez que controla interrupciones externas y comunicaciones con servidores de software, permitiendo control manual, externo e interno.

En comparación, usando el código generado de la HAL como base, estas mismas funcionalidades requerirían alrededor de 10 000 líneas; y aunque la comparación de cantidad de líneas no es necesariamente rigurosa, no es la única muestra de facilidad de uso de la ST-LIB. Abstracción de punteros de memoria, capacidad de uso de las librerías de C++ std como parámetros, estructuras simplificadas y con una nomenclatura que no requiere de conocimiento experto para poder utilizarla.

Además de ser una muestra de como usar en mayor escala la ST-LIB, también es un test de funcionalidad de la librería de alta magnitud.

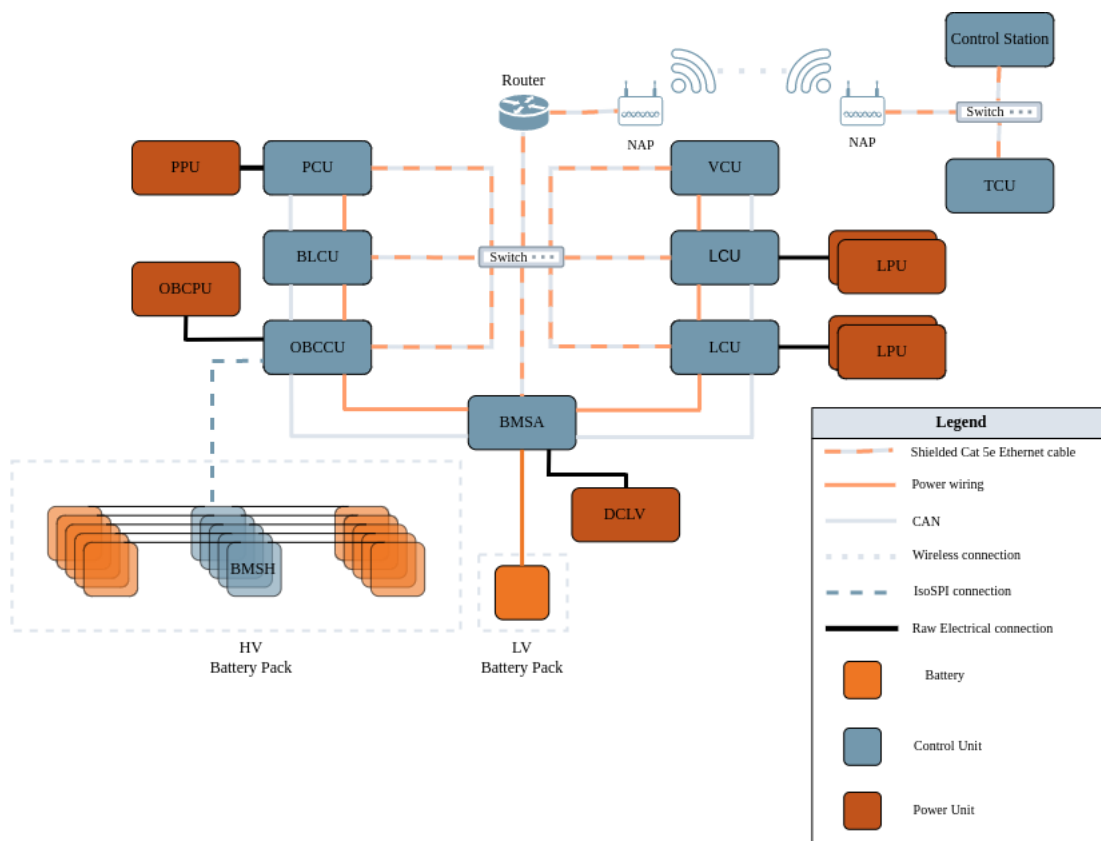


Diagrama 46: Imagen extraída de el FDD de HyperloopUPV H8 que demuestra como funciona la infraestructura de comunicaciones en el sistema

Todas las placas de la estructura (que poseen microcontrolador) usan la librería ST-LIB en su última versión, y deben coordinarse correctamente entre ellas, con el servidor, y con las placas sin controlador para representar el sistema informático de un vehículo. Son la mayor prueba del funcionamiento de la librería, y se coordinan usando varios protocolos de comunicación; especialmente Ethernet.

Explicar el funcionamiento completo de este sistema ocuparía demasiado espacio y no entra dentro del alcance de este documento; pero en resumen cada placa funciona de forma autónoma y tiene la capacidad de recibir órdenes desde otra placa (principalmente la VCU) para cambiar su comportamiento dentro de un rango marcado por los requisitos de uso.

La VCU puede pedir a la LCU que comience a levitar o deje de hacerlo; pero dependiendo de su estado será la LCU la que decida si se puede levitar y responda a la VCU si su orden es viable. La mayoría de placas tienen implementado al menos un predictor propio para controlar la toma de decisiones, y son estos predictores combinados con las protecciones de la librería las que permiten producir un producto seguro, fiable y funcional en una estampa de tiempo tan pequeña y con recursos tan limitados.

Los módulos de la librería que a más estrés son sometidos son las comunicaciones, las protecciones, las PWM, y el control de interrupciones; que son a la vez los sistemas más complicados de poner a funcionar y más propensos a fallos dentro del mercado de microcontroladores para sistemas eléctricos.

La primera prueba y la más importante es el correcto funcionamiento de la máquina de estados global. La máquina de estados global representa el estado del vehículo completo, y solo puede encontrarse en iniciando, durmiendo, operativo y fallo. Para encontrarse en durmiendo, operativo o fallo, todos los estados de cada placa deben ser durmiendo, operativo o fallo; respectivamente. Solo puede estar iniciando por un máximo de diez segundos desde que se encendió el sistema, y este estado implica que al menos una placa está iniciando y ninguna está en fallo.

Solamente el sistema de la máquina de estados general requiere el uso de la mayoría de los módulos de la librería para mantener la consistencia de un estado general. ErrorHandler, StateMachine, TCP, SNTP, notification, ProtectionManager y Time son especialmente sensibles. Para que la máquina de estados general funcione correctamente, todos estos módulos deben funcionar también.

Los procedimientos seguidos para la prueba de la máquina de estados general son sencillos. Con el servidor ya en marcha, todas las placas deben encenderse en una estampa de tiempo similar (para ello usan la misma línea de alimentación controlada por varias piezas de hardware o son alimentadas por otras placas), y las placas deben ser capaces de resolver si todas las comunicaciones están en correcto funcionamiento en 2 segundos.

Tras la correcta inicialización de la máquina de estados general, esta se queda esperando hasta que el servidor de luz verde y entre en modo operativo. Si cualquiera de las placas detecta una situación para la que no está programada; como conexiones Ethernet no funcionales, simplemente propaga el estado **FAULT** a todas las placas a las que haya

conseguido conectarse y todas consiguen llevar el estado general a fallo.

7.3. Testing automático

Durante la fase de desarrollo y parte de la fase de testeo se preparó una herramienta especial para poder hacer pruebas continuas a la librería conocida como la ATP (*Automatic Testing Platform*). La ATP se divide en una pieza de hardware especializada que emula múltiples señales utilizadas en casos de uso reales denominada SHUTP y una OrangePi 5 con ubuntu como *RTOS* que hace de servidor para recibir los test a probar y enterarse de las *pull-request* abiertas en GitHub de las cuales obtener el código, a la cual simplemente denominaremos OrangePi.

La SHUTP se compone de una placa con 400 pin outs²¹ que hace de montura para dos núcleos emuladoras debajo y un microcontrolador arriba (*que puede estar en una nucleo o en una placa personalizada*). Las nucleos emuladoras tienen un código fijo que puede recibir señales del microcontrolador a testear y que dadas las señales que reciben tratarán de simular una batería de periféricos para los que la librería fue planteada.

Las señales que las emuladoras reciban servirán para indicar que periféricos se quieren simular y para imitar las señales que reciben los periféricos seleccionados. Estos periféricos incluyen: Todos los protocolos de comunicación, un motor lineal con su encoder, sensores de temperatura y presión, válvulas, pwm, y otras nucleos con sus propios procesos.

En cuanto al microcontrolador a testear, está pensado para recibir el código a probar desde bootloader, que deberá incluir todas las comunicaciones necesarias para el debugging y las señales para activar las *nucleos* emuladoras en los periféricos que se deseen probar en susodicho código. La SHUTP se diseñó para poder ser usada tanto en La ATP como fuera de esta, pero para el caso de la ATP esta tendrá montada una núcleo en la conexión de la micro a testear y todas las comunicaciones con la OrangePi se harán a partir del bus PCAN²².

Para la OrangePi se prepararon múltiples Gits en GitHub que contienen la librería, los códigos a probar, la batería de tests que estos deben pasar, y el propio código de la OrangePi.

El código de la OrangePi es un servidor de Python hecho en Flask que está diseñado para revisar los Gits que contienen los códigos a probar en busca de cambios y *pull-requests* usando los WebHooks de GitHub. Una vez GitHub de un aviso de cambio en uno de estos Gits (o alternativamente un cliente se conecta al servidor OrangePi) el servidor creara un hilo de ejecución que descargara la rama a probar y ejecutara todos los tests comunicándose con la SHUTP, luego terminara su ejecución y devolverá la información al servidor OrangePi que dependiendo de si las pruebas dieron positivo o no aceptara el

²¹Los pin outs son unas conexiones de circuitería que permiten conectar un componente específico de una placa con cualquier componente externo

²²PCAN es un protocolo de comunicación productor/consumidor donde los primeros responden a las peticiones del segundo.

PR o dará información sobre los errores que sucedieron. En el caso de múltiples peticiones simplemente usa una cola FIFO²³.

Los tests que deben pasar el código de la placa son comprobaciones que indican el orden en el que deben de suceder las cosas y comprueban a través del bus Ethernet si el resultado ha sido el esperado. Estos indican que código se ha de subir a la SHUTP, que peticiones se le han de mandar a dicho código a través de Ethernet, y dado estas peticiones que resultados le debe devolver la núcleo y en que margen de tiempo. Estos tests están hechos en Python y son ejecutados en orden alfabético y de forma atómica. Si un test falla las pruebas continúan y simplemente se registra el fallo. Si era una prueba de cliente, también se le da la opción de interrumpir los tests y se le avisa nada más se da el fallo.

El código a probar debe ser uno que usando la ST-LIB emule una ejecución de un caso de uso real, este adaptado para activar las núcleos emuladoras de la SHUTP en los modos deseados para hacer las pruebas, y comunique continuamente los datos necesarios para que los tests puedan juzgar si la ejecución funciona como se espera y los resultados son los correctos. Este será el código que se suba junto a la librería a través del Bootloader. La separación de test y código a testear es por comodidad. En lugar de hacer un gran test con múltiples datos para cada código, se separa en varios tests que comprueben distintas funcionalidades de forma modular, haciendo más fácil añadir nuevos tests en el futuro sin afectar a los ya existentes.

En cuanto a los tests en sí, se creó una librería de python que permitía emular el funcionamiento de una placa en el nivel de comunicación y guardaba la estructura de pines de un microcontrolador para poder hacer con facilidad peticiones a la SHUTP desde el test, como leer el valor del Pin PC6 cuando el Pin PA5 está en alta y cuando está en baja, y obtener los tiempos de diferencia desde que se envía la petición hasta que llega el cambio a la OrangePi.

La razón para hacer tests automáticos es principalmente para no tener que repetir los tests cada vez que se actualice la librería. Al tratar de abstraer algo de tan bajo nivel como el código de los microcontroladores, tratando de mantener tanto la eficiencia como no comprometer la dificultad de uso de la librería, es muy fácil cometer errores. No tener en cuenta un comportamiento específico, desconocer algunas propiedades específicas de una funcionalidad que se desea abstraer, o incluso intentar añadir cambios que no son posibles o su costo en otro ámbito lo hace irrazonable.

Debido a esto, la librería requiere de mantenimiento y cambio continuo, y al tratarse de un código tan cercano al metal el orden de las acciones y la velocidad a la que se ejecutan puede ser la diferencia entre un funcionamiento correcto o un cortocircuito que pueda dañar toda la infraestructura. Cada vez que se añadía un módulo este problema aumentaba exponencialmente, y los problemas de escalabilidad a la hora de trabajar con la librería surgen mucho más pronto por todas las razones mencionadas.

²³cola First In First Out, el primero que entra es el primero que sale, el tipo de cola más común en el día a día

Los tests automáticos tienen como objetivo entonces evitar que se repitan errores, y asegurarse de que los nuevos cambios hechos son efectivamente funcionales en la práctica, y no ha sufrido la librería un fallo humano. Al fin y al cabo, si se quiere hacer una librería fácil de usar, es obligatorio que sea consistente, pues lo más difícil de usar en el mundo de la programación es un código con errores.

Al final, los tests automáticos permitieron no solo encontrar errores de la ST-LIB, sino también errores de la HAL e incluso de diseño de otros sistemas los microcontroladores; como el Bootloader, que tenía un error en reinicio de la placa que ponía los pines de Bootloader a tierra en lugar de desconectados, lo cual se traduce en un cortocircuito.

Se consiguió parchear estos errores desde el nivel de la librería o usando archivos modificados a los que la ST-LIB referencia, aumentando la consistencia incluso por encima de la de un código bien estructurado de menor nivel. Por ello, aunque preparar estos tests tuviera un gran coste temporal al comienzo del proceso, a lo largo del año consiguió pagar su coste temporal ofreciendo información de depurado de muy alta importancia.

8. Discusión del proyecto

El proyecto ST-LIB fue el primer intento de un equipo creciente de crear una librería completa que abstraiese los casos de uso del código de microcontroladores que se creaba para hacer funcionar el vehículo hyperloop H8 Kenos.

Al principio de año se tuvo muchas ideas sobre como podría funcionar, y el concepto de la librería mutó múltiples veces antes de llegar al punto en el que se encuentra ahora.

Estos cambios continuos han dejado algunos vestigios en la estructura de la librería, algunos pequeños y otros más destacados. El uso excesivo de la HAL es una de las cosas de la que la librería peca, aunque en parte es justificable por la estricta franja de tiempo en la que se tuvo que desarrollar para llegar a la competición EHW.

Al abusar demasiado de la HAL somete al código a funcionar con las estructuras de esta, y si se desea crear estructuras propias como es el caso; se termina teniendo un esquema de estructuras anidadas muy enrevesado, con mapas que contienen estructuras de la ST-LIB con estructuras de la HAL dentro de ellas, cada una de estas requiriendo su inicialización en un instante del código concreto que complica el uso para nuevos usuarios.

La HAL es una herramienta útil, pero como todo tiene sus casos de uso y limitaciones. Adicionalmente, muchas veces es posible entrar dentro del código de esta y revisar como funciona, pudiendo extraer las líneas de código necesarias directamente. Esto haría posible la creación de una ST-LIB que trabajara directamente desde el código máquina usando a la HAL como ejemplo, sin embargo, el desarrollo de una librería así tomaría mucho más tiempo de desarrollar. La opción más factible habría sido usar la ST-LIB, pero de forma más sabia y sobre todo evitando o reescribiendo todos los métodos que usaran el reloj artificial de la HAL `uwtick` (un punto de fallo recurrente durante el desarrollo que no aportaba ninguna utilidad imprescindible).

Otro problema es la estructura de inscripción general de la librería. A lo largo del año se fueron implementando cada vez más acciones en tiempo de compilación, y cuando la estructura de la librería ya estaba formada, se vio que muchas de las acciones de la ST-LIB a la hora de registrar periféricos y activarlos podrían haberse hecho en tiempo de compilación; usando templates y otras estructuras que generaran código.

La estructura de como se resolvería esto no llevo a ser concretada, pero de hacerse no eliminaría la necesidad de que el usuario tuviese que modificar documentos en el proyecto como el Runes y Pins y mejoraría mucho el tiempo de inicio del microcontrolador (este sería completamente dependiente de los límites de tiempo en comunicaciones).

El último fallo de la librería es de estructura. La idea de separar por capas de complejidad la librería ha funcionado perfectamente y es la forma lógica de organizarla; pero se ha fallado en abstraer la configuración de estas capas correctamente. Lo correcto sería tener un documento `options.hpp` que fuese opcional y permitiera sobrescribir la configuración de los módulos de la librería sin modificarla.

Parte de esta configuración está en el Runes, parte en los *headers*, otra en la inscrip-

ción de periféricos y la configuración del reloj global está en las variables de entorno del compilador²⁴.

Una vez se tiene conocimiento de la estructura de la librería, esto no es un problema excesivo, pero puede requerir que cada microcontrolador tenga una versión de la librería propia con las variables modificadas y es una barrera de entrada para nuevos usuarios, lo cual para el propósito de la librería es negativo. Otra opción sería añadirlo en el **STLIB::start()** en lugar de usar un documento, pero hay demasiada configuración e incluso usando estructuras esta se volvería una línea de código demasiado aparatosa.

²⁴Vease apartado 4.2 IDE

9. Conclusión

La librería ST-LIB tiene multitud de funcionalidades que ofrecen una gran ayuda a la hora de sacar un proyecto a corto y medio plazo, y es una buena forma de introducir al usuario al mundo de los microcontroladores.

Sin embargo, como se puede ver en el apartado de perfilado, esta librería no es perfecta, y puede encontrarse con una penalización de eficiencia respecto a las capacidades de LL. Cualquier librería especializada para un proyecto a largo plazo puede ofrecer un aumento de eficiencia razonable; sobre todo a nivel de espacio de código y tiempo de inicio. Además, esta librería se puede crear para que ofrezca un uso más acomodado para el caso de uso específico de la empresa, aunque siga requiriendo de usuarios expertos en materia.

El objetivo de la librería no es tratar de renovar completamente el mercado de la programación en microcontroladores, sino ofrecer una alternativa a proyectos de ámbito didáctico y a empresas iniciándose en el mundo de la programación de bajo nivel. Un punto intermedio entre los extremos que son Arduino y la librería HAL. Que permita hacer un programa de gran tamaño, competente a nivel de optimización, y sin requerir años de trabajo para resolverlo completamente.

En ese ámbito, se puede considerar a la librería como un éxito completo. A pesar de esto, es cierto que a la librería, en el momento que se escribe este documento, le faltan módulos que un usuario introduciéndose en el mundo de los microcontroladores podría echar en falta.

Esta librería, al fin y al cabo, es un trabajo conjunto y un proyecto de código abierto aún en construcción; y aunque se puede esperar que en un futuro se añadan las utilidades faltantes, es muy probable que no contenga todas las utilidades requeridas por el usuario. Así pues, la librería no debe servir a este solo como herramienta; sino también como ejemplo.

Al final, todo proyecto creciente llegará a un punto en el que sus componentes e infraestructuras sobre las que se construyó queden obsoletas. Maquinaria mecánica, estructuras de organización de trabajo, tecnologías completas de comunicación y medida; y por supuesto las librerías utilizadas en la programación del código.

En ese entonces el equipo que dirija el proyecto tendrá dos opciones. Actualizar los componentes, que sería ir creando versiones propias de la ST-LIB para nuestro caso; o cambiarlos, que para un equipo que trabaje en microcontroladores, su única opción realista sería crear a través de la librería LL una librería especializada usando únicamente las estructuras de C.

Tratar de seguir nuestros pasos y crearla con la librería HAL usando estructuras de C++ no sería una opción razonable, pues se encontrarían con los mismos problemas de eficiencia y requisitos de la HAL que la ST-LIB encontró, y que en parte le dio forma.

En ese momento, este documento y la librería ST-LIB aún puede servir al usuario, como ejemplo de los problemas que se pueden encontrar al desarrollar una librería de dicha envergadura.

Bibliografía

- [1] “Librería st-lib.” [Online]. Available: <https://github.com/HyperloopUPV-H8/ST-LIB.git>
- [2] “Pagina portada de la EHW.” [Online]. Available: <https://hyperloopweek.com/>
- [3] “Instalador de git.” [Online]. Available: <https://git-scm.com/download>
- [4] “Como instalar git.” [Online]. Available: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>
- [5] “template-project.” [Online]. Available: <https://github.com/HyperloopUPV-H8/template-project.git>
- [6] “Pagina de instalación del stm32CubeIDE.” [Online]. Available: <https://www.st.com/en/development-tools/stm32cubeide.html>
- [7] “idea básica del can-fd.” [Online]. Available: <https://www.can-cia.org/can-knowledge/can/can-fd/>
- [8] “Wiki del encoder dentro del github de la st-lib.” [Online]. Available: <https://github.com/HyperloopUPV-H8/ST-LIB/wiki/Encoder-Sensor>
- [9] “Código de la placa tcu del tubo atlas de hyperloop upv h8.” [Online]. Available: <https://github.com/HyperloopUPV-H8/TCU-H8.git>
- [10] “Como usar cordic en stm32.” [Online]. Available: https://www.st.com/resource/en/application_note/an5325-how-to-use-the-cordic-to-perform-mathematical-functions-on-stm32-mcus-stmicroelectronics.pdf
- [11] “Organización hyperloop upv.” [Online]. Available: <https://github.com/HyperloopUPV-H8>